

HEAPS, PRIORITY QUEUES AND HEAPSORT. HUFFMAN CODING

Chapter 5 - 5.5 of GT

DISCLAIMER: *Report an error / typo to the author so that he resolves it.*
These notes are NOT to be distributed outside of NJIT without the expressly written permission of A. Gerbessiotis

Priority Queues

Introduction

We have already defined the Abstract Data Type that is known as priority queue in Chapter 1.

Definition 1 (ADT priority queue). The ADT known as priority queue (PQ) consists of items with associated priorities (numeric values) as part of its data model that support the following operations. We call an item and its associated priority an element.

- $\text{Insert}(S, x)$ that inserts element x into PQ S ,
- $\text{RemoveMax}(S)$ (occasionally also $\text{FindMax}(S)$) that removes (finds) the element with the highest priority (that is the element with the maximum priority value),

It is also possible to define a PQ T with symmetric operations $\text{FindMin}(T)$, $\text{Insert}(T, x)$, and $\text{RemoveMin}(T)$.

A Priority Queue (PQ) defined in the Introduction supported operations Insert and $\text{ExtractHighPriority}$. We use the term RemoveMax for the latter from now on. We will thus associate a HighPriority with a large (MAX) value.

Definition 2. Binary Heap A (binary) heap is an efficient implementation of the data model underlying a priority queue. In other words, a heap is an efficient data structure solution for a PQ.

Definition 3 (Efficient Data Structures). An efficient data structure is a data model realization of an ADT that allows operations defined on the ADT to be efficient.

Priority Queues

Introduction

The data structure known as **(binary) heap** allows a PQ to be implemented efficiently, that is all the operations Insert and RemoveMax to have running time that is $O(\lg n)$, where n is the number of elements in the PQ. (A FindMax(S) operation has running time $\Theta(1)$.) It is now quite clear why a PQ is not used to implement a Stack or a Queue. Insertion in a Stack or a Queue and Deletion are constant time operations. Even if we use a Binary Heap to implement a priority queue such operations involving priority queue operations would require $O(\lg n)$ time, which is worse.

Remark 1. We note that the notion of a **(binary) heap** as defined here has nothing to do with the same term used in compilers to mean a memory area in which objects are created and occasionally destroyed using operations such as malloc, free, new, or delete.

Note 1. A (binary) heap as it will be formally defined is an array and not a binary tree; in the latter case one should use the term binary heap tree.

Moreover note that some textbooks implement and utilize not a (binary) heap but a binary heap tree and call it a heap. The latter is a binary tree that has some Heap related property (to be defined as a MaxHeap property). Such a structure is space inefficient. The binary heap that will be introduced is space efficient though.

Binary Heaps

Introduction

From now on we shall call a binary heap, a heap, dropping the binary.

Definition 4 (Binary MaxHeap). A (binary) MaxHeap A is a data structure that utilizes an array that has the following properties.

- **Elements of H .** Each element of a heap has some numeric value attached to it; it also carries additional information. We refer to this additional information as item. In the context of a priority queue implemented by a MaxHeap, we might call the numeric value a priority.
- **Array implementation:** $A[0..length(A) - 1]$. We store elements in an array A . The length of the array $l(A) = length(A)$ refers to the maximum capacity of the heap. For brevity $l(A)$ is an alias for $length(A)$.
- **Number of elements of the heap is $heapsize(A) = hs(A)$.** A has $heapsize(A)$ (for brevity $hs(A)$) denoting the number of elements of A that form a heap. Those elements are $A[0..heapsize(A) - 1]$. Moreover $heapsize(A) \leq length(A)$. Using an $l(A)$ and an $hs(A)$, A can contain no heap elements or be full of heap elements. In the former case $hs(A) = 0$ and in the latter $hs(A) = l(A)$. We can forfeit the use of **heapsize** if instead we maintain a pointer/reference to the **last** element of a heap.
- **MaxHeap-Order property or just MaxHeap property.** Every element of heap A satisfies the MAXHEAP property that states

$$\forall i < heapsize(A) \quad A[i] \geq A[2i + 1] \quad \text{AND} \quad \forall i < heapsize(A) \quad A[i] \geq A[2i + 2]$$

as long as $2i + 1$ and $2i + 2$ host elements of the heap; if not the corresponding condition is true by default. Note the two properties are equivalent to $A[\lfloor (i - 1)/2 \rfloor] \geq A[i]$.

Binary MaxHeaps

An alternative view

For the remainder we only use the numeric value (priority) and ignore the item part of an element. The numeric value of element i would thus be $A[i]$ rather than say a $A[i].priority$. Several times n will denote the length of A or the number of elements of the heap. Do not get confused.

We may define similarly a MinHeap. The only differentiation is that it defines a MINHEAP property analogously (symmetrically). (Then we have $A[\lfloor (i-1)/2 \rfloor] \leq A[i]$ for every i defined.)

In the remainder we will be dealing with a Max-Heap (i.e. a heap that maintains the MaxHeap-Order property), although the results we will obtain can be easily translated into equivalent results for a Min-Heap.

Another equivalent representation (view) of a (binary) heap is by a complete binary tree whose all levels but the lowest (with highest depth) are filled with elements and in the last level elements of the heap are filling the tree left to right. The nodes of this incomplete last level complete binary tree are numbered top-to-bottom, left-to-right starting from the root (first element with label 1). The numbered of a node indicate the index of array H mapped to that vertex. A parent labeled i has at most two children labeled $2i+1$ and $2i+2$ if they both exist (this is true for an array whose first element is $A[0]$). We call this view (heap) tree view. The equivalence between the array A that is a heap and its heap-tree view is immediate.

Sometimes it is also called a **complete binary tree view** or it is referred to as MinTree or MaxTree (compare to MinHeap and MaxHeap). Given the tree view of the array-based (binary) heap,

- a. **parent** (i) returns $\lfloor (i-1)/2 \rfloor$
- b. **left** (i) returns $2 * i + 1$;
- c. **right** (i) returns $2 * i + 2$;

Such operations in C/C++ can be implemented with shift operations efficiently. A floor is not needed in such languages as integer division is floor by default. The **height** of a tree node, is the size (in edges) of the longest **downward** path from the node to a descendant leaf of the tree. or equivalently, it is the height of the tree minus the depth of the node. The height of a tree is the height of the root. A heap with n elements has a tree view of height $\lfloor \lg n \rfloor$. A heap with n elements has a tree view with $\lceil \lg(n+1) \rceil$ levels (the root is at level 1 and so on).

Binary (Max) heaps

Example

Example 1. An array of length 8 is given as shown below.

$A = [70, 50, 60, 30, 20, 40, _, _]$
 $l(A) = 8$.

Is it true that A is a MaxHeap with $\text{heapsize}(A) = 6$?

$i=0$ $A[0] \geq A[1]$ is true since
 $70 \geq 50$
and
 $i=0$ $A[0] \geq A[2]$ is true since $\implies i=0$ has MaxHeapProperty
 $70 \geq 60$

 $i=1$ $A[1] \geq A[3]$ is true since
 $50 \geq 30$
and
 $i=1$ $A[1] \geq A[4]$ is true since $\implies i=1$ has MaxHeapProperty
 $50 \geq 20$

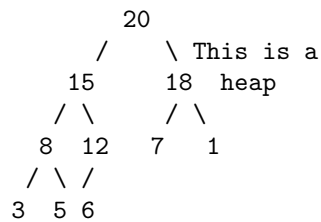
 $i=2$ $A[2] \geq A[5]$ is true since
 $60 \geq 40$
and
 $i=2$ $A[2] \geq A[6]$ is true since $\implies i=2$ has MaxHeapProperty
 $60 \geq (\text{by default}) \ 6 > \text{heapsize}(A) - 1$.

Binary (Max) heaps

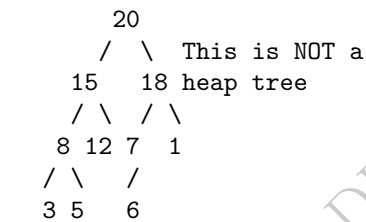
Example

Example 2. Array A next is a Max-Heap. Figure (1) shows the array as a heap-tree that has the MaxHeap-property. Figure (2) is not a heap as nodes indexed 10 and 11 are not filled with elements (the node indexed 9 holds a 5, and the node index 12 holds a 6: there is a gap, i.e. nodes in last level not filled left to right). In Figure (3) in brackets we indicate the index of a node that is the same as the index of that element within array A .

0 1 2 3 4 5 6 7 8 9
 $A = [20 \ 15 \ 18 \ 8 \ 12 \ 7 \ 1 \ 3 \ 5 \ 6]$



Figure(1)



Figure(2)

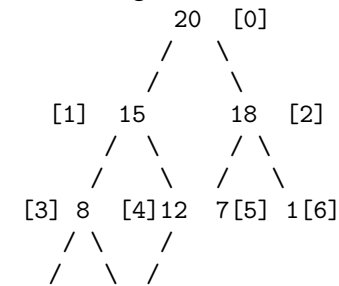


Figure (3)

Note: a $[.]$ indicates the index of A containing the corresponding element

Note: An element is its numeric value

Max heaps Operations

Some operations defined on heaps are the following ones. We assume that the elements are comparable and thus a total order is defined on them.

Definition 5. The following operations are defined on a binary heap that is a MaxHeap.

- **InsertMaxHeap**(A, x). An element x is inserted into the heap.
- **RemoveMaxHeap** (A). It removes the element with the maximum priority of the heap; after the removal the heap-property is maintained for the remainder. For a min-heap we can equivalently define **RemoveMin** (A).
- **DownHeap**(A, i). The operation is also known as Shift, Sift, Fix, or Heapify. It preserves the heap-order property at the element indexed i , if the heap-order property is not valid. The precondition is that the left and right subtree views of that element have the heap-order property; the tree view rooted at i may not have the heap-order property and the operation DownHeap resolves it. Depending on the heap we use we can call it MAXDownHeap or MinDownHeap.
- **BuildMaxHeap**(A). Given an unordered array A , a (binary) heap is built in place.
- **HeapSort** (A). Given an (unordered) array A it sorts A in place.

Max heaps

InsertMaxHeap

Theorem 1 (InsertMaxHeap). *Operation InsertMaxHeap into a binary max heap of n keys has running time $O(\lg n)$. It is implemented through function InsertMaxH.*

Proof. InsertMaxH implements InsertMaxHeap that inserts an elements into binary heap that might represent also a priority queue. Heap is represented by binary max heap A and we assume $h(A) < l(A)$ i.e. insertion is possible. It first creates space for one more elements in A and inserts k one position past the last element of the heap call it i . If there is a violation of the MaxHeap property it will involve index q (newly inserted key) and index $i/2$ (C/C++ implies a floor). If it is so (a violation) keys at i and $i/2$ get swapped: swapping might go higher up and eventually reaches the top $i = 1$ and stops (or equivalently the 'root' of the tree view and stops).

```
InsertMaxH(A,k)    // parent(i) see prior definition
1. heapsize(A) = heapsize(A)+1;
2. i=heapsize(A)-1;
3. while ((i>0) and A[parent(i)] < key) {
4.   A[i]=A[parent(i)];
5.   i=parent(i);
6. }
7. A[i]=key;
```

The running time is $O(\lg n)$ for inserting a single element in a heap or priority queue already containing n keys. An insert operation might go all the way up or stop by exiting the while loop of lines 3-5 immediately.

“Height” of Tree View. The **height** of a tree node, is the size (in edges) of the longest **downward** path from the node to a descendant leaf of the tree. A heap with n elements has a tree view of height $\lfloor \lg n \rfloor$. This lends to the logarithmic term of the time bound. ■

Max heaps

InsertMaxHeap example

Example 3 (Insertion example).

$$l(A)=8, \quad \text{hs}(A)=6$$
$$A = \begin{bmatrix} 70 & 60 & 50 & 30 & 20 & 40 & \dots & \dots \end{bmatrix}$$

•

.

```
.Insert(A,10)
```

$$l(A)=8 \quad .hs(A)=7$$

•

$$A = \begin{bmatrix} 70 & 60 & 50 & 30 & 20 & 40 & 10 & \dots \end{bmatrix}$$

•

.

```
.Insert(A,80
```

$$l(A)=8 \quad .hs(A)=8$$

•

$$A = \begin{bmatrix} 70 & 60 & 50 & 30 & 20 & 40 & 10 & 80 \end{bmatrix}$$

```

      70
     /  \
    60    50
   /  \  /  \
  30  20 40

```

```

      70
     /  \
    60    50
   /  \  /  \
  30   20 40  10

```

$$A = \begin{bmatrix} 70 & 60 & 50 & 80 & 20 & 40 & 10 & 30 \end{bmatrix}$$
$$A = \begin{bmatrix} 70 & 60 & 50 & 80 & 20 & 40 & 10 & 30 \end{bmatrix}$$
[illegible]
$$A = \begin{bmatrix} 70 & 80 & 50 & 60 & 20 & 40 & 10 & 30 \end{bmatrix}$$
$$A = \begin{bmatrix} 80 & 70 & 50 & 60 & 20 & 40 & 10 & 30 \end{bmatrix}$$

Max heaps

RemoveMaxHeap

RemoveMaxH implements RemoveMaxHeap that removes from the MaxHeap its largest (maximum value) element. The Max in a MAXHEAP is at index $i = 0$ and it is $A[0]$. A FindMaxH(A) operation in this context is trivial; it returns $A[0]$. The operation is trivial to implement; the max is $A[0]$ and at the start of the array. It can be removed easily. The problem is that the heap must remain a heap. No gaps are allowed and thus something needs to be put into $A[0]$. The easy choice is the last element of the heap. But the last element will replace the maximum, and we might cause a violation of the MaxHeap property at $i = 0$. We can fix such a violation using a function MAXDownHeap(A,i) that fixes the MaxHeap property, if it is in violation at any node indexed i of A . In our set up $i = 0$ and thus we call MAXDownHeap(A,0). However fixing the MaxHeap property at $i = 0$ or i in general can cause a cascade violation either at $2i + 1$ or $2i + 2$. And that cascade might continue to the end of the Heap. Using the tree view, a violation at the root might (or might cause) a violation at one of its two children which might propagate all the way down to a leaf. The pseudo-code of MAXDownHeap(A,i) is given following the code RemoveMaxH.

```
RemoveMaxH(A)
1. if heap-size(A) < 1
    return("Error. No heap elements");
2. max=A[0];
3. A[0]=A[heap-size(A)-1];
4. heap-size(A) = heap-size(A)-1;
5. MAXDownHeap(A,0);
6. return(max);

MAXDownHeap(A,i) //[i] vs [2i+1] vs [2i+2]
// potential violation of MaxHeap property at i
0. if i > n/2 return;
1. maxindx=max(A[i], A[left(i)], A[right(i)] );
2. if (maxindx != i) {
3.     swap (A[i],A[maxindx]);
4.     MAXDownHeap(A,maxindx); //recursively continue
}
```

DRAFT

$A[i]$
/
 $A[2i+1]$ $A[2i+2]$

Theorem 2 (RemoveMaxHeap). *Operation RemoveMaxHeap on a binary max heap of n keys has running time $O(\lg n)$. It is implemented through function RemoveMaxH.*

Proof. When MAXDownHeap is called on the element indexed i the implicit assumption is that the subtree views rooted at $\text{left}(i)$ and $\text{right}(i)$ are heaps; $A[i]$ however may violate the heap-order property. The purpose of DownHeap is to slide down this key so that the subtree view rooted at i becomes a heap.

Running time and Comparison count of MAXDownHeap. In line 1 of MAXDownHeap we find the maximum of 3 keys: this requires 2 comparisons. This is the fundamental step of lines 1-3. We might recur or might not. Potentially we might recur to the extreme of the binary heap or in its tree view we might reach a “leaf”. Thus the number of recursive calls of Line 4 starting initially with i is the height of the node indexed i in the array or in the tree view. If $i = 1$ we are talking about the height of the tree of the tree view i.e. $\lfloor \lg n \rfloor$. Every time we enter MAXDownHeap we perform two comparisons in line 1, thus the maximum comparison count is $2\lfloor \lg n \rfloor \leq 2\lg n$ in the worst-case. Note that in the last recursive call the node indexed i is a leaf, and by default it has the MaxHeap-Order property. Line 0 detects such a situation and exits before a comparison is performed. Time-wise this is $O(\lg n)$: the time bound include not only the comparisons of line 1 but also the activity of lines 2 and 3.

Running time and comparison count of RemoveMaxH. The running time of RemoveMaxH is the running time of Line 5 involving MAXDownHeap since lines 1-4 and line 6 are constant time. Thus running time is $O(\lg n)$ as well. Note that in terms of comparisons, RemoveMaxH requires as many as MAXDownHeap i.e. $\leq 2\lg n$. ■

Max heaps

MaxDownHeap

A MaxDownHeap operation applied to the element at index i , it also involves the elements at index $2i + 1$ and $2i + 2$. In the examples that follow the element on which a MaxDownHeap operation is performed is indicated with a star next to it: in the array view the star is on top of the position of the node and in the tree view the star is on its right.

Corollary 1. One application of MaxDownHeap at the element index i takes $\Theta(1)$ time or 2 comparisons to complete.

On application of MaxDownHeap at i fixes a violation of the MaxHeapProperty at i . It might subsequently cause a violation of the MaxHeapProperty at $2i + 1$ or $2i + 2$ or not. A MaxDownHeap operation starting at the element at index i might propagate further down the array (or the tree of the tree view). Eventually it stops for elements (leaves of the tree view) that have the MaxHeapProperty by default.

Corollary 2. An application of MaxDownHeap at the element index i that propagates until its completion requires time $O(\lg n)$ and at most $2 \lg n$ comparisons.

Proof. This is a direct consequence of the height of the tree view of a MaxHeap capable array accommodating n elements. In other words the sequence,

$$0 \rightarrow 1 \rightarrow 3 \rightarrow 7 \rightarrow \dots 2^{k+1} - 1$$

involving the i of MaxDownHeap(i) propagating downwards in the worst case (starting with $i = 0$) has the trace shown above and it (max i) exceeds $n - 1$ i.e. $2^{k+1} - 1 > n - 1$ for $k = \lg n - 1$. The number of MaxDownHeap operations is then $k + 1 = \lg n$, each involving 2 comparisons. ■

The operation MaxDownHeap is also known as Sift, or Shift, or Heapify.

RemoveMaxHeap example

Example 4 (RemoveMaxH example).

```

l(A)=8,    hs(A)=8    a * indication an application of MaxDownHeap
A = [ 80 70 50 60 20 40 10 30 ]

      |
      | RemoveMax
      |
A = [    70 50 60 20 40 10 30 ]
      |
      | hs(A)= 7
      |
      *
A = [ 30 70 50 60 20 40 10 -- ]
      *|
A = [ 70 30 50 60 20 40 10 -- ]
      *|
A = [ 70 60 50 30 20 40 10 -- ]
      |
A = [ 70 60 50 30 20 40 10 -- ]

```

Diagram illustrating the removal of the maximum element (80) from a max heap and the subsequent restructuring of the array A to maintain the heap property.

The initial heap structure (A = [80 70 50 60 20 40 10 30]) is shown as a tree with root 80. The root is swapped with the last element (30), and the new root (30) is bubbled up to maintain the heap property.

The resulting heap structure (A = [70 30 50 60 20 40 10 --]) is shown as a tree with root 70. The root is swapped with the last element (10), and the new root (10) is bubbled up to maintain the heap property.

The final heap structure (A = [70 60 50 30 20 40 10 --]) is shown as a tree with root 70. The root is swapped with the last element (10), and the new root (10) is bubbled up to maintain the heap property.

MaxHeaps BuildMaxHeap

<pre>//Iterative BuildMaxHeap BuildMaxHeap(A,n) 1. heapsize(A)=l(A); 2. for(i=floor((l(A)-1)/2);i>=0;i--){ 3. MAXDownHeap(A,i); }</pre>	<pre>//Recursive BuildMaxHeap RBMaxHeap(A,n) RBMaxHeap(A,i) 1. heapsize(A)=l(A); 1.if(left(i)<heapsize(A)) 2. RBMaxHeap(A,0); 2. RBMaxHeap(A,left(i)); 3.if(right(i)<heapsize(A)) 4. RBMaxHeap(A,right(A)); 5.MAXDownHeap(A,i);</pre>
---	---

BuildMaxH(A,n). The operation BuildMaxH(A,n) builds a MaxHeap out of an arbitrary array of n elements. At the conclusion of BuildMaxHeap A will be a MaxHeap. Currently before line 1 it might not be (actually it is not). Thus the first action is to potentially set $hs(A) = n$ and the start fixing violations of the MaxHeap property from the extremes of the binary heap array A . Note that the leaves of the tree view of A have the MaxHeap property by default as the node indexed i has no nodes indexed $2i + 1$ and $2i + 2$ for a leaf. Thus we only need to check for violations for the remaining roughly $n/2$ elements of A (be reminded in a binary tree, number of internal nodes is one less than the number of leaves). This the reason that the for loop starts the way it is.

A naive analysis of BuildMaxHeap observes that the number of iterations is $n/2$ and the running time of line 3 i.e. of a MaxDownHeap(A,i) is at most the time of the most time consuming MaxDownHeap(A,1) which is $O(\lg n)$. Thus one might conclude that the running of BuildMaxHeap is indeed $O(n \lg n)$. It is so but the bound is NOT tight enough. We can do a better and more careful analysis. The reason for that is that the MaxDownHeap(A,i) operation does not ALWAYS operate on the node indexed $i = 1$. In fact only once for the $\approx n/2 - 1$ invocations does it involve $i = 1$. Consider a complete binary with n nodes. The last level has m leaves, the second last $m/2$ internal node and the root is 1 node. The total is $2m - 1$. (This also says that the number of leaves is $(n + 1)/2$.)

MaxHeaps

BuildMaxHeap simplified analysis

Theorem 3 (BuildMaxHeap). *Operation BuildMaxHeap builds a binary MaxHeap out of n input keys in running time $\Theta(n)$ or in at most $2n$ comparisons. It is implemented through function BuildMaxHeap.*

Proof. (Simplified) Analysis. A tighter bound follows from the simple observation that only the root might require $\lg n$ iterations for MAXDownHeap; more than half of the nodes are leaves and thus the cost of MAXDownHeap is zero, for one quarter or more the cost of MAXDownHeap is one iteration per node and so on. The number of nodes at height h is $\lceil n/2^{h+1} \rceil$. Each of them will perform at most h iterations (calls/operations) of DownHeap, each one of them computing the maximum of 3 elements i.e. 2 comparisons. Thus in total the number of iterations of DownHeap is at most $hn/2^{h+1}$ and twice as much is the number of comparisons. The total running time is then the order of the total number of MAXDownHeap iterations which is at most

$$\sum_{h=0}^{\lg n} hn/2^{h+1} = n/2 \cdot \sum_{h=0}^{\lg n} h(1/2)^h \leq n/2 \cdot \sum_{h=0}^{\infty} h(1/2)^h \leq n/2 \cdot 2 = n$$

where we used the fact (Appendix) that for $x < 1$, $\sum_{i=0}^{\infty} ix^i \leq x/(1-x)^2$ and for $i = h$, $x = 1/2$ we get $\sum_{h=0}^{\infty} h(1/2)^h \leq 2$. Each Line 1 of MAXDownHeap involves at most 2 comparisons. Thus the number of MAXDownHeap function-calls is at most n , the number of comparisons performed (2 per function call) is at most $2n$ and the running of BuildMaxHeap is thus $O(n)$. ■

MaxHeaps BuildMaxHeap detailed analysis

Proof. (Detailed analysis of BuildMaxHeap(A,n).)

Bottom level with $m/2$ leaves. The nodes have by default the MaxHeap property. No comparisons.

Next to bottom level with $m/4$ internal nodes. On each node we might have (including recursive calls) two applications of MaxDownHeap on a node of that level and one of the bottom level; the latter exits through line 0 without comparisons. $2 \cdot 1$ comparisons per node for a total of $m/4 \cdot (2 \cdot 1)$.

2nd to bottom level with $m/8$ internal nodes. Each one of them requires ≤ 3 MaxDownHeap operations but only $2 \cdot 2$ comparisons maximum per node from the top two MaxDownHeap.

Adding up all contribution of MaxDownHeap in line 3 of BuildMaxHeap.

For the comparison contribution $C(n)$ of MaxDownHeap in all recursive calls for all iterations of line 3 of BuildMaxHeap we have the following expression. (Note that $2m - 1 = n$ and we use the Appendix.)

$$\begin{aligned} C(n) &= \sum_{i=1}^{i=\lg m} (m/2^i) \times 2 \cdot i = 2m \sum_{i=1}^{i=\lg m} (1/2)^i \times i \leq 2m \sum_{i=0}^{i=\lg m} (1/2)^i \times i \\ &= 2m(2 - (2(\lg m + 1) + 2)/2^{\lg m+1}) = 4m - (2 \lg m + 4) = 2n + 2 - 2 \lg m - 4 \leq 2n. \end{aligned}$$

In line 3 the previously finite sum is upper bounded by an infinite sum. The addition of $i = 0$ has no effect as the corresponding term is 0. Each Line 1 of DownHeap performs at most 2 comparisons. Thus the number of MaxDownHeap function-calls is at most $n + 1$, the number of comparisons performed is at most $2n$ and the running of Build-MaxHeap is thus $O(n)$.

$C(n)$ and $T(n)$ of BuildMaxHeap. If the comparison count of all MaxDownHeap operations is $C(n) \leq 2n$ then the running time of BuildMaxHeap is dominated by the MaxDownHeap operations and it is also $T(n) = O(n)$. In fact $T(n) = \Theta(n)$, as we need $\Theta(n)$ time just to read the n input keys. ■

Exercise 1. Analyze the running time of RBMaxHeap. Derive $T(h) = 2T(h - 1) + h$ (i.e. use height information rather than size information) for its running time.

BuildMaxHeap Example

Example 5.

$l(A)=4$, $hs(A)=4?$ a * indication an application of MaxDownHeap
 *
 $A = [10 \ 50 \ 30 \ 140]$ STEP SKIPPED $i=3$ has MaxHeapProperty by default
 *
 $A = [10 \ 50 \ 30 \ 140]$ STEP SKIPPED $i=2$ has MaxHeapProperty by default
 *
 $A = [10 \ 50 \ 30 \ 140]$ 10 10 10* 140 140
 *| / \ / \ / \ / \ / \ / \
 $A = [10 \ 140 \ 30 \ 50]$ 50* 30-->140 30-->140 30-->10* 30-->50 30
 * / / / / /
 $A = [10 \ 140 \ 30 \ 50]$ 140 50* 50 50 10*
 *|
 $A = [140 \ 10 \ 30 \ 50]$
 *|
 $A = [140 \ 50 \ 30 \ 10]$

Max heaps HeapSort

We can do 'selection sort' using a max heap utilizing a max rather than a min. The resulting algorithm is HeapSort.

Theorem 4 (HeapSort). *Operation HeapSort sorts n input keys in running time $\Theta(n \lg n)$. It is implemented through function. Its performance can be summarized as $C(n) \leq 2n \lg n + 2n$ and $T(n) = \Theta(n \lg n)$. Space (additional to input) is $S(n) = \Theta(1)$. The algorithm sorts in-place but is not stable. (Note that for this discussion n is a power of two, or as needed otherwise for us to be able not to use floors or ceilings.)*

Proof. **Heapsort** utilizes BuildMaxHeap to first build a heap from the n input keys. The key with the maximum value is stored in the top of the heap (root of heap tree or key in index position 1 of array A), and thus it can be extracted from the heap and placed in its correct position in the output sequence (the last position). This is achieved by exchanging $A[0]$ and $A[i]$. This exchange, may violate the heap property for elements $A[0 \dots i]$ (note that the extraction of max makes the heap of size one less). It is obvious however that the "children" of $A[0]$ in the tree view have the heap-property since the rest of the heap was untouched by the exchange. A call to MAXDownHeap will maintain the heap property for $A[0]$ and the resulting $A[0..i-1]$ becomes a heap. We continue similarly with the extraction of the current maximum key from the heap $A[0..i-1]$.

```
HeapSort(A,n)           // A is an unordered array of n keys
1. Build-Max-Heap(A);    // A has become a heap
2. for(i=n-1;i>=1;i--){  // A[0..i] is a heap A[i+1..n-1] is a sorted subarray
3.     swap(A[0], A[i]);  // Remove Max and place it into A[i],move A[i] to A[0]
    // A[0..i-1] may not be a heap, A[i..n-1] is a sorted subarray
4.     heapsize(A) = heapsize(A)-1; //one key was removed from heap;adjust size
5.     MAXDownHeap(A,0);    //Root may not have Heap Property; Run DownHeap
}
```

Running time and Comparison count of HeapSort. Step 1 requires time $O(n)$. The loop of lines 2-5 is executed $n - 1$ times and each call to MAXDownHeap takes $O(\lg n)$ steps. The total running time is thus $O(n \lg n)$ in the worst case. Moreover the comparison count of HeapSort can also be estimated. Based on prior discussion of BuildMaxHeap line 1 requires at most $2n$ comparisons. Line number is a MAXDownHeap executed $n - 1$ times (upper bound n times) and in each time the MAXDownHeap uses index $i = 0$. The number of comparisons involved would be roughly at most $2 \lg n$ per line 5 invocation. For $n - 1$ invocations we get a total of $2n \lg n$. Thus the comparison count of HeapSort is at most approximately $C(n) \leq 2n \lg n + 2n$. ($\lg n$ might not be an integer.) ■

HeapSort Example

The operation BuildMaxHeap for this example A was shown earlier in detail. The bar (pipe symbol $|$) below indicates the boundary between the input that has not been processed yet (on the left side of the $|$) and the partial output already formed (on the right side of the $|$).

Example 6 (HeapSort example).

A = [10 50 30 140] Input	Stability :	A = [10_1 10_2]
BuildMaxHeap		BuildMaxHeap
A = [140 50 30 10]		A = [10_1 10_2]
A = [10 50 30 140]		A = [10_2 10_1]
A = [50 10 30 140]		A = [10_2 10_1]
A = [30 10 50 140]		10s have been swapped
A = [10 30 50 140]		
A = [10 30 50 140] Output		

Introduction: strings and words

In the remainder we deal with yet another application of priority queues, that of text compression. It is also known as data compression.

Definition 6 (Alphabet). An alphabet is a set in which a number of characters are defined. (Or an alphabet is a set of characters defined and used to express in written form a language.)

We have the English (character) alphabet, the Greek alphabet, and so on.

Definition 7 (Word, String). A string is a sequence of characters from some alphabet A . A word is a sequence of characters from some alphabet A that has some meaning. A word is a string but a string might not be a word.

Definition 8 (Length of Word or String). It is the number of its characters. For a word w or string s its length is denoted by $|w|$ and $|s|$ respectively.

Definition 9 (Prefix and Suffix). Let $w = w_1w_2 \dots w_n$ be a string of length n over some alphabet A , where $w_i \in A$, for all $1 \leq i \leq n$. A string $w_1w_2 \dots w_k$, $1 \leq k \leq n$ is known as a prefix of w (of length k). A string $w_{n-k}w_{n-k+1} \dots w_n$, $0 \leq k < n$ is known as a suffix of w (of length $k + 1$).

The characters of an alphabet are represented in the memory of a computer using some special coding. We thus refer to the code of a character of an alphabet. Such codes can be byte aligned codes or bit aligned codes.

Compression

Introduction: codes

Definition 10 (Code). The code of a character of an Alphabet is a unique bit sequence that represents the character uniquely from all of the other characters of the Alphabet.

The ASCII character set defines the codes not only of the English characters (upper-case and lower case) but also of the ten digits, punctuation marks and other diacritics. There is also the Unicode and UTF-8. They are all binary codes; the code of a character is a bit sequence (of 0s and 1s).

Definition 11 (Byte aligned code). A byte aligned code is a binary code where the codes of individual characters respect byte boundaries. Thus every code of every character is one or more byte.

Every code of a character of the original ASCII character set was 7bit and fits into a 1B to the point that we say an ASCII character is 1B. Unicode characters have codes that are 1B, 2B, 3B, etc. In UTF-8 we represent characters from ASCII or Unicode transparently with the minimal number of byte needed. Thus an ASCII character has a UTF-8 code that is 1B and it is its ASCII code, but UTF-8 can represent other, more than 1000000, Unicode characters in 2B, 3B, 4B, as needed.

Definition 12 (Bit aligned code). A bit aligned code is a binary code where the codes of individual characters do not respect byte boundaries and thus it is a variable number of bit per character. Thus there are codes for some character that are neither 1B or 2B or etc in length.

Example 7. A bit aligned code for the alphabet of characters $\{A, C, G, T\}$ (that are being use to represent the human genomoe) is 00, 01, 10, 11 respectively. The code for each character is 2bit and thus four characters (code) can fit into a byte. Another bit aligned code is 0, 10, 110, and 111 respectively for A, C, G and T.

Example 8. In ASCII, the code for the character (digit) 0 is 48 in decimal, or 30 (0x30) in hexadecimal, or **0b0110000** in binary. Likewise digit 1 is **0b00110001**. In UNICODE this digit 0 would be encoded 0x0030 in hexadecimal i.e. two bytes are being used, one of which is zero and the other contains the ASCII value of character 0. In UTF-8 the code for digit 0 is the ASCII code.

The ASCII codes for 0 and 1 sound counter-intuitive.

Definition 13 (Number of bit for a fixed bit aligned code). Since n bit allow for the representation of 2^n distinct values, an alphabet that contains $|A|$ characters requires $\approx \lg |A|$ bit per character if a fixed number of bit are to be used to code (encode) the characters of the alphabet.

Definition 14 (Encode). When we convert character digit 0 into its ASCII code **0110000** we say we encode digit 0 into its ASCII equivalent (including the denary 48, and hexadecimal 0x30 view.)

Definition 15 (Decode). When we convert code **0110000** from ASCII into the digit 0 we say we decode the ASCII code into its alphabet form. (This also includes decoding 48 or 0x30.)

Definition 16 (Text, length, size). For a text X let $n = n(X)$ be the length of its alphabet A and $N = c(X)$ be the length of the text X . We may enumerate the characters of A as c_i , where $i = 1, \dots, n$. The size of the text can be defined in B or bit. Thus

$$\text{size}_{\text{bit}}(x) = 8 \cdot \text{size}_{\text{B}}(x) = 8$$

Definition 17 (Text, frequency and probability). For a text as defined in Definition 16, each character c_i of A has some frequency of occurrence f_i such that $\sum_{c_i \in A} f_i = N$. The "probability" of occurrence of character c_i is its fraction of occurrence and defined as $p_i = f_i/N$.

Example 9. Let $X = \text{ACGTAAAGTCACACAA}$. Then $\Sigma = \{A, C, G, T\}$. (We use Σ to denote the alphabet, as A is a character symbol.) Then $n(X) = 4$ and $c(X) = 16$. Thus $n = 4$ and $N = 16$. Furthermore

$$f_A = N/2, \quad f_C = N/4, \quad f_G = f_T = N/8$$

Moreover,

$$p_A = 1/2, \quad p_C = 1/4, \quad p_G = p_T = 1/8$$

Note that $\sum f_i = N$ and $\sum p_i = 1$.

If ASCII is used for the coding of the text X , then $\text{size}_B(X) = 16B = NB$ and $\text{size}_{\text{bit}}(X) = 128\text{bit} = 8N\text{bit}$.

The ASCII code used for the coding is a byte aligned code. We utilize next a bit aligned code for encoding the characters (and text) of X .

Example 10. For Example 9 we encode the text X (i.e. its characters) using a bit aligned code as follows.

$$A : 00, C : 01, G : 10, T : 11$$

We can pack four characters into a byte this way. Then $size_B(X) = 16/4B = N/4B$ and $size_{bit}(X) = 128/4bit = 2Nbit$.

Definition 18 (Compression ratio). The latter encoding of Example 10 achieves better space utilization. We denote the compression ratio as the following ratio.

$$s = \text{compression factor} = \frac{\text{Original size}}{\text{Chosen Encoded size}}$$

Thus the **compression factor** = s can be expressed as s_B or s_{bit} . In either case $s_B = N/(N/4) = s_{bit} = 8N/2N = 4$.

Definition 19. The entropy e of a text X is defined as follows.

$$e = - \sum_i p_i \lg p_i = - \sum_i \frac{f_i}{N} \lg \frac{f_i}{N}$$

The entropy of a text indicates how much information is incorporated (embedded) into the text. It subsequently indicates a lower bound on the number of bit we need to use to encode the characters of the text.

Example 11. The entropy of the text of example 9 is

$$e = -\frac{1}{2} \lg \frac{1}{2} - \frac{1}{4} \lg \frac{1}{4} - \frac{1}{8} \lg \frac{1}{8} - \frac{1}{8} \lg \frac{1}{8} = 1.75$$

Compression

Entropy lower bound

Theorem 5. *Using codes such as the ones described earlier (bit aligned) a text X with length $N = c(X)$ with entropy e can be encoded to such a degree such that*

$$size_{bit} \geq eN \text{bit}.$$

Corollary 3. The (bit) lower bound eN is achievable if the inverses $1/p_i$ are powers of two.

Example 12. Since $1/p_A = 2^1, 1/p_C = 2^2, 1/p_G = 1/p_T = 2^3$, there is an encoding of A, C, G, T that has

$$size_{bit} = eN = 1.75 \cdot 16 = 28bit.$$

For example 12 the prefix codes

$$A : 0, C : 10, G : 110, T : 111$$

satisfy the lower bound. More on the following page(s).

Definition 20 (Prefix codes). A prefix code for the characters of an alphabet A are codes assigned to the characters such as any code IS NOT a prefix of another code.

Example 13. The codes of Example 10 are prefix codes.

$A : 00, C : 01, G : 10, T : 11$

Another set of prefix codes are as follows.

$A : 0, C : 10, G : 110, T : 111$

Example 14. The codes with encoding E , such that $E(A) = 0, E(C) = 1$, etc i.e.

$A : 0, C : 1, G : 01, T : 10$

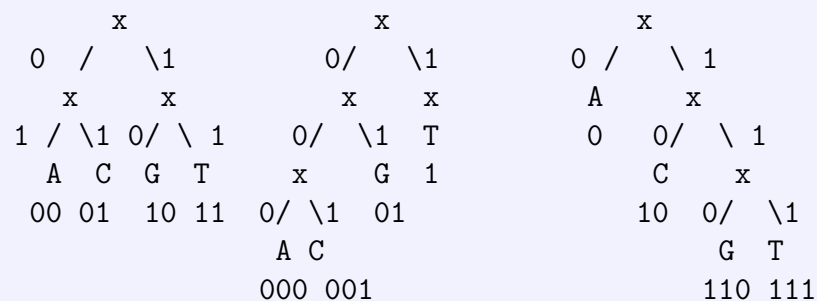
are not prefix codes. Let $X = ACACCACA$. After we encode X using E into X' we get into $X' = 01011010$. If we attempt to decode X' absence of additional knowledge we might end up with text $X_1 = GGTT$ or $X_2 = GACTT$ or other variations!

Finding Prefix codes: binary trees

In the following definition A represents both an alphabet and also a character within the alphabet. Bear this in mind!

Theorem 6 (A full binary tree $T(A, B)$ is (leads to) a prefix code). *Let A be an alphabet of n characters and let $B = \{0, 1\}$. A prefix code for the n characters of A can be generated over B quite easily by making the n characters leaves to a binary tree $T(A, B)$ that is a full binary tree. The code for a character c_i is an encoding of the path from the root of the binary tree to the leaf corresponding to c_i . A left branch can be mapped to a 0 and right branch (edge) to a 1. The same alphabet can have multiple prefix codes (and the number of them for n characters is a Catalan number.)*

Example 15. For the Alphabet containing A, C, G, T we can find easily three sets of prefix codes.



The rightmost binary tree generates the code below

$A : 0, C : 10, G : 110, T : 111$

Corollary 4. The prefix codes of Example 15 corresponding to the rightmost binary tree, offer for the example text of Example 9 an encoding of its characters that satisfy the lower bound of Corollary 3 and the bound of Example 12.

Proof. For the prefix codes of

$$A : 0, C : 10, G : 110, T : 111$$

the length $d(c_i)$ of each character code is respectively $d(c_A) = d_A = 1bit$, $d(c_C) = d_C = 2bit$, $d(c_G) = d_G = 3bit$, $d(c_T) = d_T = 3bit$, and thus under this encoding, the encoded X has bit size

$$size_{bit} = N/2 \cdot d_A + N/4 \cdot d_C + N/8 \cdot d_G + N/8 \cdot d_T = N/2 \cdot 1 + N/4 \cdot 2 + N/8 \cdot 3 + N/8 \cdot 3 = 1.75N = 1.75 \cdot 16 = 28bit$$

This mean that this chosen prefix code satisfies Corollary 3 and Theorem 5. ■

Question 1. How can we build OPTIMAL PREFIX CODES optimally and what is the optimality criterion?

Answer. By using a method due to Huffman. The optimality criterion is the minimization of the function

$$size_{bit}(X') = \sum_i f_i d_i = \sum_i f_i d(c_i).$$

That is, find an encoding E that generates optimal prefix codes thus achieving the minimum

$$\min_E size_{bit}(X')$$

where X is a text, X' is the encoding of text X under the optimal E that minimizes $size_{bit}(X')$. In the rest of the formulae above f_i is the frequency in X of character i (also denoted by c_i or $c(i)$) and d_i is the number of bit of the bit aligned prefix code assigned to character c_i by E that minimizes $size_{bit}(X')$.

Step 1 (Optimal). An OPTIMAL prefix code utilizes the frequencies of the characters of the alphabet A of text X .

Step 2 (Prefix Code). The objective is to find an optimal PREFIX CODE. Finding a prefix code involves generating multiple binary trees having leaves the characters of alphabet A .

Step 3 (Optimal Prefix Code). In order to generate an OPTIMAL PREFIX CODE we use Huffman's greedy algorithm that builds the binary tree from which we generate the optimal encoding E that achieves the $\min_E size_{bit}(X')$.

Step 4 (Optimal Huffman Implementation). Huffman implementation of step 3 can be optimized in terms of its computational performance (rather than the E generated) by utilizing a Binary MIN heap that maintains the characters of X and their associated frequencies. ■

Problem 1 (*Data Compression: first definition*).

Input: The input (X, A) consists of a text X composed of characters from some alphabet A .

Output: The output is $(X', B); E$,

where the text X has been encoded by E into text X' whose characters are drawn from some alternative alphabet B . E is picked so that compression is achieved i.e. $size_{bit}(X') < size_{bit}(X)$. The mapping of every character of A into a word of characters of B is known as encoding and let it be denoted by E . The ratio $size_{bit}(X)/size_{bit}(X')$ is the compression factor availed by the encoding E on (the characters A of) X .

Output $(X', B); E$ **that satisfies** $E : A \rightarrow B^+, \quad (X, A) \xrightarrow{E} (X', B)$

Text or Data Compression Encoding

Note 2 (Input and Output files). If X is associated with a file containing it, then (X', B) ; E is also associated with a file that contains not only X' the encoded text X but also the encoding E itself or the inverse E' of E .

Earlier we used the definition of the compression factor through the expression.

$$\text{size}_{\text{bit}}(X)/\text{size}_{\text{bit}}(X')$$

We could have used instead

$$|X|/|X'|$$

that is being used in practice. We did not do so nor will do so because we do not want to get into the details on how to store E or E' along with the encoded text in X' .

Note 3. The inverse E' of the encoding E must also map uniquely a word of B that appears in X' into a character of A .

It helps that E generates prefix codes and thus a word of B in X' won't be a prefix of another word w_1 of X' .

Definition 21 (*Encoding*). An encoding E from A to B is a function from A to B^+ that is

$$E : A \rightarrow B^+.$$

The term B^+ means one or more characters of B , i.e. a word of B .

Definition 22 (*Code*). The code of character $x \in A$ is $E(x) \in B^+$ under encoding E as defined above.

Relation (function) E' is defined as the inverse of E and allows us to retrieve or generate x from $E(x)$.

Problem 2 (*Data Compression: first definition*).

Input: $(X', B); E$ or alternatively $(X', B); E'$.

The input consists of an encoded text X' composed of characters from some alphabet B and the encoding E that resulted in (X', B) , or E' itself, the inverse of E .

Output: The output is either X or alternatively X_1 such that $X_1 \approx X$.

When the output is X the compression/decompression scheme is lossless; when the output is X_1 such that $X_1 \approx X$ the compression scheme is lossy.

If E' is not given, starting with $(X', B); E$ we compute E' from E and then generate X or (X, A) in the case of lossless compression or X_1 or (X_1, A) in the case of lossy compression. If E' is given the first part of computing E' is obviously skipped.

Definition 23 (*Decoding*). A decoding is also defined which is a relation E' from B^+ to A . In fact it is a relation from $\text{Range}(A) \subset B^+$ to A . It is only needed to be defined for the characters of A that are in X .

$$E' : \text{Range}(A) \rightarrow A.$$

Example 16. Consider for example the case where A are the digit 0, 1, 2, 3. One possible encoding is for $x \in A$ to have $E(x) = \text{ASCII}(x)$. Then $E(x) \in B^+$, where $B = \{0, 1\}$. For example $E(0) = 00110000$, $E(1) = 00110001$, $E(2) = 00110010$, and $E(3) = 00110011$. For this specific case $E(x) \in B^8$ or $E(x) \in B^7$ since we can represent a digit with 7bit in ASCII (stored in practice in one byte consisting of eight bit). B^7 or B^8 means words of length 7 or 8 whose characters are from $B = \{0, 1\}$. Note that the bit sequence for $E(0), \dots, E(3)$ requires one byte for storage. Thus under this assumption $|X| = |X'|$. Function (or relation) E' is defined so that for example $E'(00110011) = 3$.

Example 17. One might argue that a better encoding E_1 is the following one $E_1(0) = 00$, $E_1(1) = 01$, $E_1(2) = 10$, $E_1(3) = 11$. Furthermore if we pack four two-bit words into a byte, we can generate an X' such that $|X|/|X'| = 4$.

Text or Data Compression

Compression and Decompression

(This is a formal repetition of previous definitions. You may skip it.) In data or text compression the first step is to determine efficiently an efficient encoding E that would result in $|X'| < |X|$. Under E the code of a character of A that appears in X would be determined using characters of the target alphabet B . These codes must be unique and uniquely identify the characters of A . This is the **encoding** phase. In X' the characters of A would then be stored through their respective codes thus $c \in A$ would be mapped and stored in X' in the form $E(c)$, where $E(c) \in B^+$. This is the **compression phase**. More often that not both phases are known as encoding phase or compression phase.

In order to recover X from X' we need to be able first to compute E' that recovers the codes of characters of A from the words of B stored in X' . It is thus imperative that X' stores information about E' or information about E from which E' can be derived (quickly and efficiently). This is the **decoding** phase. We also refer to it as the **decompression phase**.

If the size (in characters, in byte, or in bit) of X' is smaller than that of X we say that we have compressed X into X' . Otherwise we just say we have encoded X in the form of X' .

Definition 24 (Compression). It describes a solution to the data compression problem. Starting with (X, A) we compute E and generate (X', B) .

$$E : A \rightarrow B^+, \quad (X, A) \xrightarrow{E} (X', B)$$

We can simplify this into $X \xrightarrow{E} X'$, to the detriment of rigor.

Definition 25 (Decompression). It describes a solution to the inverse of the data compression problem. Starting with $(X', B); E$ we compute E' and generate X or (X, A) .

$$(X', B); E, \quad E' : \text{Range}(A) \rightarrow A, \quad (X', B); E \xrightarrow{E'} X \quad \text{or} \quad (X', B); E \xrightarrow{E'} (X, A).$$

We can simplify this into $X' \xrightarrow{E'} X$, to the detriment of rigor.

Lossless and Lossy compression

Overview

Definition 26 (Lossy Decompression). It describes a solution to the inverse of the data compression problem. Starting with $(X', B); E$ we compute E' and generate X or (X, A) .

$$(X', B); E, \quad E' : \text{Range}(A) \rightarrow A, \quad (X', B); E \xrightarrow{E'} X'' \quad \text{or} \quad (X', B); E \xrightarrow{E'} (X'', A), \quad \text{where} \quad X'' \approx X.$$

We can simplify this into $X' \xrightarrow{E'} X''$, to the detriment of rigor. In other words through decompression the best we can hope for is to retrieve an approximation of X rather than X itself. Moreover X' might be X''' such that $X''' \approx X'$.

Definition 27 (Lossless compression). A compression and decompression phase that generates X' and then X respectively is a lossless compression or more formally a lossless compression scheme (or algorithm).

Definition 28 (Lossy compression). A compression and decompression phase that generates X' and then X'' , where $X'' \approx X$, is a lossy compression or more formally a lossy compression scheme (or technique or algorithm).

Thus lossless or lossy is usually determined by the decompression phase. A lossy algorithm is also known as noisy.

JPEG, MPEG are lossy compression algorithms. GIF is a lossless one based on the Lempel-Ziv-Welch (LZW) algorithm. PNG is also lossless (and for sometime was used as a replacement for GIF).

Example 18. Let $X = ACTGAACA$ with $\mathcal{A} = \{A, C, G, T\}$. Then $|X| = 8B$ or $|X| = 64bit$. We define $B = \{0, 1\}$. One encoding $E_1 : \mathcal{A} \rightarrow B^+$ is as follows $E_1(A) = 0$, $E_1(C) = 10$, $E_1(G) = 110$, $E_1(T) = 111$. Then $X' = 01011111000100$. Then, $|X'| = 2B$ or $|X'| = 14bit$. The compression factor is $8/2$ or $64/14$. X' does not store E . If E is given separately, can we retrieve X from X' and E ? The answer is YES.

Example 19. Let $X = ACTGAACA$ with $\mathcal{A} = \{A, C, G, T\}$. Then $|X| = 8B$ or $|X| = 64bit$. We define $B = \{0, 1\}$. Another encoding $E_2 : \mathcal{A} \rightarrow B^+$ is as follows $E_2(A) = 0$, $E_2(C) = 1$, $E_2(G) = 10$, $E_2(T) = 01$. Then $X' = 0101100010$. Then, $|X'| = 2B$ or $|X'| = 10bit$. The compression factor is $8/2$ or $64/10$. X' does not store E . If E is given separately, can we retrieve X from X' and E ? The answer is NO. In X' we do not know if the prefix 01 is an A followed by C or it is a T . Both $TTGAAG$ or $ACTGAACA$ can be generated from X' and E . We do not have such a problem with E_1 because the E_1 codes are prefix codes: no code is a prefix of another code.

Codes

Prefix, Bit-aligned and Byte-aligned

(A more formal definition of prefix codes.)

Definition 29 (*Prefix codes*). An encoding E of alphabet A generates a prefix code $E(x)\forall x \in X$, if no $E(x)$ is a prefix of another $E(z)$, $\forall x \in A, \forall z \in A, x \neq z$.

Example 20. For E_2 as defined earlier, $E_2(A)$ is a prefix of $E_2(T)$ and also $E_2(C)$ is a prefix of $E_2(G)$. Thus $E_2(x)\forall x \in \{A, C, G, T\}$ is not a prefix code. Moreover $E_1(x)\forall x \in \{A, C, G, T\}$ is a prefix code.

DRAFT

Prefix Codes

Huffman codes for Huffman compression

The primary method that will be presented in this section for text (or data) compression is Huffman's algorithm. It is also known as Huffman's encoding, or Huffman's compression method, or Huffman's coding. It yields what are known as Huffman codes.

Definition 30 (Huffman codes). The Huffman codes of a text X is an encoding E of the characters of text X that form alphabet A that generate bit-aligned prefix codes. $(X, A); E$ then leads to a lossless compression algorithm.

Definition 31 (Efficient Huffman code generation). An efficient Huffman code generation uses a priority queue to generate Huffman codes.

When we discuss in the remainder and beyond Huffman codes, we mean an efficient generation of Huffman codes that utilize a priority queue.

Huffman codes result in space savings of 20% to 90% depending on the structure of text X that will be compressed. Nowadays it is not very often that anyone is using Huffman compression to compress a text or strings of characters. Average savings observed are of 3 bits per character thus using 5 bits or so for character encoding. This is not good enough any more. Lempel Ziv (LZ78) uses only 4.2 bits on the average, and LZ77 a bit less. The UNIX `compress` function uses approximately 3.31 bits and `gzip` close to 2.53. And `bzip2` uses 2.23 bits per character which implies close to a 4:1 compression. However instead of doing character encoding using Huffman coding, we can do word encoding, and Huffman can exhibit substantially better performance to the level of 2.95 bits per word. The important property of Huffman/prefix codes is that decoding is quite simple. Since no codeword is a prefix of another one, that codeword at the start of a string or a file is unambiguous and can be removed from the coded file; repeating this process decodes the rest of the text uniquely. No 'separator' is needed to separate adjacent codes.

Definition 32 (Prefix codes from full binary trees). A prefix code is generated from a full binary tree $T(A, B)$ as explained in Theorem 6 using input A and B .

Definition 33 (Huffman codes are optimal prefix code). Let X be a text over alphabet A and $B = \{0, 1\}$. A Huffman code is an optimal prefix code. An optimal prefix code is generated from an optimal full binary tree $\mathcal{T}$ that is also known as the Huffman tree $\mathcal{T}$. An optimal full binary tree $\mathcal{T}$ is the $T(A, B)$ generated by Theorem 6 using input A and B that minimizes

$$\sum_{a \in A} d_{T(A, B)}(a) f_{T(A, B)}(a)$$

where $f_{T(A, B)}(a)$ is the frequency of character a in X (and also in A) and $d_{T(A, B)}(a)$ is the path length (in edges) from the root of $T(A, B)$ to the leaf that maps to character a . In other words $\mathcal{T}$ satisfies,

$$\mathcal{T} : B(\mathcal{T}) = \min_{T(A, B)} \sum_{a \in A} d_{T(A, B)}(a) f_{T(A, B)}(a)$$

Huffman codes

Prefix codes from binary trees

Whereas $d(a)$ is the path length from the root to a specific leaf that is a character, the path itself as explained in Theorem 6 defines the code associated with the character in question.

It is possible that there are more than one $\mathcal{T}$ $B(\mathcal{T})$. Any one of those trees can be used.

Remark 2. Implementation-wise, A is of cardinality 256 as any byte value may be mapped to a “character”.

Huffman’s algorithm for encoding and decoding a text is a greedy algorithm that during its course of execution makes a decision that is the best possible decision at the time This is the greedy principle in algorithm design. In order for $\mathcal{T}$ to be determined the greedy principle uses information maintained by $f_{T(A,B)}(a)$.

DRAFT

Huffman codes

Huffman compression phase

For a given text X and set of characters A that are used in X , their associated frequencies are being used by Huffman's compression method to determine the Huffman codes of the characters of X .

Problem 3 (Huffman Compression phase).

Input: Text X over alphabet A that is (X, A) . Frequency $f(a)$ of every character $a \in X, a \in A$.

Output: $(X', B); E$.

$$E : A \rightarrow B^+, \quad (X, A) \xrightarrow{E} (X', B)$$

For a text X and alphabet A , for every $a \in A$, the frequency $f(a)$ of a in X is determined first. Thus for all a $f(a) > 0$. Let $n = |A|$.

We then form n individual one-node trees, one per character a and associate with character a its frequency $f(a)$. Every node is also the root of its one-node tree.

Then the algorithm works bottom-up by combining two available trees into a single new one by creating a new root node for the new tree whose left child is the root of one and the right child the root of the other of the two available trees to be combined. The frequency field of the new root is the sum of the frequencies of the roots of the two constituent trees. This is also the sum of the frequencies of the leaves in the subtree rooted at that node. There is no character representation associated with the newly created root node.

The greedy principle is applied in picking two available trees: it is the two trees whose roots r_1, r_2 have the smallest frequency associated with them. An optimal realization (algorithm) for finding those two trees is by using a priority queue that maintains the roots of those trees. The priority of a root is its frequency.

Eventually, after $n - 1$ combining operations we obtain a full binary tree $\mathcal{T} = T(A, B)$ using Huffman's construction utilizing the characters' frequencies. The tree then generates a Huffman code that is an optimal prefix code. The number of bits required to encode X for a given Huffman tree $\mathcal{T}$ is $B(\mathcal{T}) = \sum_{a \in A} f_{\mathcal{T}}(a) d_{\mathcal{T}}(a)$, where $d(a)$ is the depth of character a in the tree that is equal to the number of bit for the Huffman code of a , and $f(a)$ is the frequency of character a in the text. The following pseudocode shows the steps for compression and constructing the Huffman code for a text X including generating its output X' .

Huffman compression phase

Encode

```
Encode (X, X',A)                                // Text is ACGTAAAGTCACACT
1. Read X and compute A[a].f for all a in A;    // n=4, N=16 fA:8,fC=4,fG=fT=2
2. Tr= Huffman(A[]); //Build Huffman tree Tr that minimizes fAdA+fCdC+fGdG+fTdT
3. From Tr generate for every a of A,           //dA:1 dC=2 dG:3 dT:3
   A[a].code, the Huffman code of a            // A:0 C:10 G:110 T:111
4. EncodeFile(X,A,Tr,X') // Encoded file is X' // X':010110111000 ...
5. // Somehow a representation of Tr           // 4ACGT8422
   is to be stored in X' as well !
```

DRAFT

Huffman compression phase

Huffman

The following pseudocode shows the steps of building Huffman's tree.

```
Huffman (A[ ])//Every character in A has A[.].f X[.].ch
0.  PriorityQueue Q ; // a definition/declaration of Q
1.  BuildMinHeap(Q,A);
2.  for(a=0;a<|A|;a++){
    // create single node trees
3.  Tr[a].ch= A[a].ch;
    Tr[a].f = A[a].f;
    Tr[a].left = Tr[a].right= NULL;
4.  // connect Q with the tree nodes of Tr;i.e. Q points to Tr elements
5.  }
6.  for(i=0;i<n-1;i++) {
7.      z=ALLOCATENODE();
8.      x=z.left = Extract-Min(Q); //x points to an element of Tr by step 4
9.      y=z.right = Extract-Min(Q); //y points to an element of Tr by step 4
10.     z.freq = x.freq + y.freq;
11.     z.char = -1;
12.     InsertMinHeap(Q,z); // Extract twice in 8-9; Insert once in 12
13. } // Therefore net reduction is by one
14. return(ExtractMinHeap(Q)); // Only one element in Q: the root of Tr
```

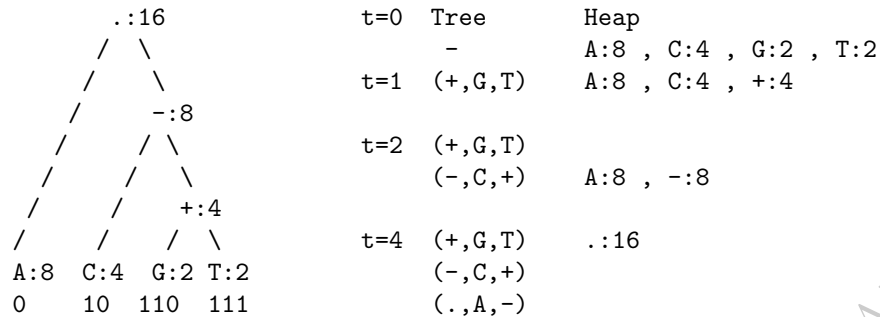
The running time of Huffman is $O(n \lg n)$, where $n = |X|$. Note that n is the number of distinct characters of S not the size of S . Regular English texts might be Megabytes in size. Yet n is no more than 100 as English texts utilize a subset of the ASCII character set. Lines 0-4 require $O(|X|) = O(n)$ time. Line 5 (BuildHeap operation) requires $O(n)$ time and lines 6-14 require $O(\lg n)$ time per iteration for a total of $O(n \lg n)$.

Huffman compression phase

Example

Text: X: ACGTAAAGTCACACT n=4 (A,C,G,T) N=16
 Freq: fA=8,fC=4,fG=fT=2
 Objective: Find codes whose bitlengths dA, dC, dG, dT
 minimize (fAdA + fCdC + fGdG +fTdT)
 How: Huffman's method (Huffman codes)

Huffman:



Output A:0 fA=8,dA=1 ,
 C:10 fC=4,dC=2 ,
 G:110 fG=2,dG=3 ,
 T:111 fT=2,dT=3 .

EncodeFile : X : ACGTAAAGTCACACT
 EncodedFile: X': 010110111000110111...

In reality X'': overhead + X'
 where X'': encodeinbin(4ACGT8422)010110111000110111
 4ACGT8422 reads: 4 ASCII chars
 A then
 C then
 G then
 T with frequencies
 8,4,2,2 respectively

Huffman decompression phase

The decoding operation traverses the tree from the root to a leaf guided by X' .

```

    DecodeFile (X', X)           // X: 4ACGT8422 010110111000
0.  Obtain Tr from X;           //   / \   A:0 C: 10 G:110 T:111
    B[]=codedtext(C);           //   /  \
    pos=0;                       //  /   \  /\
1.  pointer = root(Tr);         // A   C G T
2.  while (B[pos] != EOFFile ) { //
3.    if (B[pos] == 0)
4.      pointer = pointer.left;
5.    else
6.      pointer = pointer.right;
7.    if (isleaf(pointer) == TRUE) { // ie. pointer.ch != -1
8.      Output(pointer.ch,S);
9.      pointer=Tr;
10.   }
11. }
12. return(S);

```

The decoding procedure works in time linear to the size in bits of the encoded/compressed file C .

Remark 3. In the remainder when we refer to a Huffman tree T we will not be using the **Tr** notation. We shall call it just T .

Huffman's method

Proof of correctness: Part 1

Theorem 7 (Huffman-1). *Let A be an alphabet in which each character a has frequency $f(a)$. Let x and y be two characters of A with the lowest frequencies. Then there exists an optimal prefix code for A in which x and y have same length codewords that only differ in the last bit.*

Proof. We start with an optimal but arbitrary prefix code and modify it to obtain another optimal prefix code where x and y are maximum depth sibling in the new tree (i.e. codewords have same length and differ in last/rightmost bit).

Let in the arbitrary optimal prefix code b and c are the siblings of maximum depth. Without loss of generality let $f(b) \leq f(c)$ and also, $f(x) \leq f(y)$. Since $f(x), f(y)$ are lowest leaf frequencies, then $f(b), f(c)$ must be such that $f(x) \leq f(b)$ and $f(y) \leq f(c)$.

We obtain tree T' by exchanging in T x with b and T'' by exchanging y with c in T' .

Then we have

$$\begin{aligned} B(T) - B(T') &= \sum_c f(c)d_T(c) - \sum_c f(c)d_{T'}(c) \\ &= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b) \\ &= f(x)d_T(x) + f(b)d_T(b) - f(x)d_T(b) - f(b)d_T(x) \\ &= (f(b) - f(x))(d_T(b) - d_T(x)) \\ &\geq 0. \end{aligned}$$

Note that since $f(x)$ is minimum frequency leaf $f(b) - f(x) \geq 0$. Since b is maximum depth leaf in T $d_T(b) \geq d_T(x)$.

We can similarly prove that $B(T') - B(T'') \geq 0$. Therefore $B(T'') \leq B(T)$. Since T is optimal $B(T) \leq B(T'')$. Therefore $B(T) = B(T'')$. ■

Huffman's method

Proof of correctness: Part 2

Theorem 8 (Huffman-2). *Let T be a full binary tree representing an optimal prefix code for X with frequencies $f(a)$. Consider two sibling leaves x and y in T and let z be their parent. If z is considered a character with $f(z) = f(x) + f(y)$, the tree $T' = T - \{x, y\}$ represents an optimal prefix code for $X' = X - \{x, y\} \cup \{z\}$.*

Proof. For every c in A other than x, y we have $d_T(c) = d_{T'}(c)$. For $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$f(x)d_T(x) + f(y)d_T(y) = (f(x) + f(y))(d_{T'}(z) + 1) = f(z)d_{T'}(z) + f(x) + f(y).$$

Then, $B(T) = B(T') + f(x) + f(y)$. If $B(T')$ is not optimal then there is an optimal tree O such that $B(O) < B(T')$, and its leaves are characters in X' . Then z appears as a leaf in O . If we add x, y in O we get a prefix code whose total cost is $B(O) + f(x) + f(y) < B(T) = B(T') + f(x) + f(y)$. This, however, contradicts the minimality of $B(T)$. Therefore such an O can not exist because T' is the optimal prefix code for X' . ■

Theorem 9 (Huffman's Algorithm). *Function $Huffman$ produces an optimal prefix code.*

Proof. A direct result of Theorems Huffman-1 and Huffman-2. ■

Appendix

An equation derivation

We already know the first two derivations: the first one has n terms, the second one has infinite term. In the latter one the right-hand side is to the limit $n \rightarrow \infty$. The third one is to be derived from the second one.

$$\begin{aligned}\sum_{i=0}^{n-1} x^i &= 1 + x + \dots + x^i + \dots + x^{n-1} = \frac{x^n - 1}{x - 1} & x \neq 1 \\ \sum_{i=0}^{\infty} x^i &= 1 + x + \dots + x^i + \dots = \frac{1}{1 - x} & 0 < x < 1 \\ \sum_{i=0}^{\infty} i \cdot x^i &= 1 \cdot x^1 + 2 \cdot x^2 + \dots + i \cdot x^i + \dots = ? & 0 < x < 1\end{aligned}$$

Starting, from the second equation we work as follows: (a) take the first derivative of both sides, (b) multiply the result (both sides) by x , and for any $0 < x < 1$, we have.

$$\begin{aligned}\sum_{i=0}^{\infty} x^i &= \frac{1}{1 - x} \\ 1 + x + \dots + x^i + \dots &= \frac{1}{1 - x} \\ (1 + x + \dots + x^i + \dots)' &= \left(\frac{1}{1 - x} \right)' \\ 0 + 1 \cdot x^0 + \dots + i \cdot x^{i-1} + \dots &= \frac{1}{(1 - x)^2} \\ 0 \cdot x + 1 \cdot x^1 + \dots + i \cdot x^i + \dots &= \frac{x}{(1 - x)^2} \\ \sum_{i=0}^{\infty} i \cdot x^i &= \frac{x}{(1 - x)^2}\end{aligned}$$

If we substitute $x = 1/2$ we obtain

$$\sum_{i=0}^{\infty} i/2^i = 2$$

Appendix

An alternative analysis of RBMaxHeap

The RBMaxHeap works in a strange way. Let i be the index of the element $A[i]$ in the heap array and let $L(i)$ be the elements in the heap-array that belong to the left subtree of the i -indexed node and $R(i)$ the elements in the heap-array that belong to the right subtree of the i -indexed node. We call $l(i)$ and $r(i)$ the left, right children of the node indexed i . So $L(\cdot)$ and $R(\cdot)$ are sets of nodes, whereas $l(\cdot)$, $r(\cdot)$ are nodes.

In the call of line 2 of RBMaxHeap $i = 0$ and RBMaxHeap is entered with $i = 0$. Line 2 of RBMaxHeap builds recursively a MAX-heap out of the $L(i) = L(0)$ nodes, line 4 builds a MAX-heap of the $R(i) = R(0)$ nodes, and now AFTER $L(i) = L(0)$ and $R(i) = R(0)$ have been turned into heaps, it is possible to apply DownHeap at node $i = 0$ to build a MAX-heap out of every node of the tree.

The key to this algorithm is that DownHeap is applied AFTER $L(i)$ have formed a max-heap and after $R(i)$ have also formed a max-heap. Note that if line 5 was the first line of the code, and not the fifth this would NOT have been a CORRECT Build-MaxHeap procedure since in order to apply DownHeap at node indexed i , the left subtree and right subtrees MUST BE HEAPS. The latter is guaranteed however after the execution of lines 2 and 4 of RB-MaxHeap. Therefore the code given is a recursive Build-Heap method complementing the iterative.

Do they have the same running time? Let's analyze the running time of RBMaxHeap for a general node indexed i . Let the height of node i be h and the running time be expressed in terms of $T(h)$ (this is h not n). Then, $l(i)$ and $r(i)$ are of height one less and thus the running time of RBMaxHeap will be $T(h - 1)$ and $T(h - 1)$ respectively. The running time of DownHeap is proportional to the height of node i , i.e. it is $O(h)$. Combining all the pieces together we have

$$T(h) = 2T(h - 1) + ch$$

where we write for simplicity the $O(h)$ as ch .

Appendix

An alternative analysis of RBMaxHeap

What is the solution to this recurrence? We can use the substitution method or the iteration method to show that $T(h) = O(2^h)$. Let's try the iteration method. For base case we are going to choose $T(0) = 0$.

$$\begin{aligned} T(h) &= 2T(h-1) + ch \\ &= 2(2T(h-2) + c(h-1)) + ch \\ &= 2^2T(h-2) + 2^1c(h-1) + 2^0c(h-0) \\ &= 2^2(2T(h-3) + c(h-2)) + 2^1c(h-1) + 2^0c(h-0) \\ &= 2^3T(h-3) + 2^2c(h-2) + 2^1c(h-1) + 2^0c(h-0) \\ &= 2^hT(0) + 2^{h-1}c(h - (h-1)) + \dots + 2^2c(h-2) + 2^1c(h-1) + 2^0c(h-0) \\ &= 2^{h-1}c(h - (h-1)) + \dots + 2^i c(h-i) + \dots + 2^1c(h-1) + 2^0c(h-0) \\ &= c \sum_{i=0}^{h-1} 2^i (h-i) \\ &= c \sum_{i=0}^{h-1} 2^i h - c \sum_{i=0}^{h-1} i 2^i \\ &= ch(2^h - 1) - c \sum_{i=0}^{h-1} i 2^i \\ &= ch(2^{h+1} - 1) - c((h-1)2^{h+1} - (h)2^h + 2) \\ &= c2^{h+1} - ch + 2c. \end{aligned}$$

We used above the base case $T(0) = c$. We also identified the sum $\sum_i i 2^i$, also of the **formulae collection (available in Document 2)**. Thus $\sum_{i=0}^{h-1} i 2^i = (h-1)2^{h+1} - (h)2^h + 2$.

For a heap with n elements, the height of the root, i.e. the element indexed $i = 1$, is at most $\lg n$. Thus $T(h)$ is $O(2^{h+1}) = O(n)$. This means that the running time of RIGHT-BuildMAX-Heap is $O(n)$, just like the running time of the non-recursive variant presented in class.

Question. Why did we use in the running time analysis $T(h)$ instead of $T(n)$? For $i = 1$, how big (i.e. how many nodes does it contain) is the left-subtree $L(0)$ of the root? How big is the right subtree of the $R(0)$ of the root?

Notice

Problem Set with Solutions

These Problem Sets are not for credit. They are provided as are for your practice. Several problems are provided with solutions. These can be discussed with the instructor after a student has studied the relevant material. Some other problems are provided with either no solutions at all or a hint of a solution. They are there for your reference and will not be discussed further. For some solutions provided you might have to fill in details related to calculations or intermediate steps.

Before you try these problems: (a) Study the designated textbook, (b) use the Subject notes (typewritten notes available through the course web-page from the password-protected section C) to review your study of the designated textbook material, (c) resolve the examples done in class, (d) then practice with some of these problems PS (Problem Set), and then if you have time, (e) solve some of the problems of the designated textbook, and (f) solve more problems from the PS especially the ones with minimal solution information.

Several PS problems appeared in past exams or homeworks.

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 1.

- (a) What are the minimum and maximum numbers of elements in a heap (max- or min- heap) of height h ?
- (b) What is the height of a heap with n elements?
- (c) Show that in any subtree of a MAX-HEAP, the root of the subtree contains the largest value occurring anywhere in that subtree.

Proof. (a) Since the heap has size h , all levels but the last form a complete binary tree of height $h - 1$. Such a tree has $1 + 2 + \dots + 2^{h-1} = 2^h - 1$ nodes. A heap of height h must have at least one node (leftmost) at the next level h (that may be incomplete) because otherwise it won't be of height h but $h - 1$. Moreover the maximum number of nodes (leaves) at level h is 2^h . Therefore minimum number of nodes in a heap of height h is $2^h - 1 + 1 = 2^h$; maximum is $2^h - 1 + 2^h = 2^{h+1} - 1$.

(b) An n element heap has height h such that, by way of part (a), $2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$. Therefore by taking logarithms base 2 of both sides $h \leq \lg n < h + 1$. Height h is always an integer. By the definition of the floor function we derive that $h = \lfloor \lg n \rfloor$.

(c) Let $S(u)$ be the subtree of a heap rooted at some node u . Suppose by way of contradiction that the maximum value M stored in $S(u)$ is not at u . Let the left and right subtrees of u be $LS(u)$ and $RS(u)$. Then the maximum of $S(u)$ must be in either $LS(u)$ or $RS(u)$. Let the index of the heap node of $S(u)$ with the maximum M be m (note that if there are more than one nodes storing M we take the node with the minimum index value m). As the node indexed m cannot be node u it must be an internal node of $S(u)$ and therefore it must have a parent p . The index of the parent will be $\text{floor}(m/2)$ i.e. smaller than m . The value stored in p must also be less than M as m is the lowest indexed node storing the maximum value M and therefore the value of the parent whose index is less than m must be smaller. Therefore the MAX heap-property fails for p and m , a contradiction to the assumption that $S(u)$ is a subtree of a heap. ■

Problem Set
Problem 2.

(a) How much is the sum

$$\sum_{i=0}^{\infty} i \cdot x^i = 1 \cdot x^1 + 2 \cdot x^2 + \dots + i \cdot x^i + \dots = ? \quad 0 < x < 1$$

(b) How much is the sum $\sum_{i=0}^{\infty} i(i-1)x^i$, $0 < x < 1$? Explain.

Proof. (a) Read the Subject on Heaps and Priority Queues and one of the Appendices there. The derivation of interest is the

$$\sum_{i=0}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2}.$$

(b) Work similarly to the appendix of Subject 5 in which the formula above in (a) was derived. Start with the infinite geometric sequence.

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}.$$

Take the first derivative of both sides. We get

$$\sum_{i=0}^{\infty} i \cdot x^{i-1} = \frac{1}{(1-x)^2}.$$

Take the first derivative of both resulting sides. We get

$$\sum_{i=0}^{\infty} i(i-1) \cdot x^{i-2} = \frac{2}{(1-x)^3}.$$

We then multiply both sides with x^2 . We get

$$\sum_{i=0}^{\infty} i(i-1) \cdot x^i = \frac{2x^2}{(1-x)^3}.$$

All of the steps are ok since $0 < x < 1$. Result is shown above.

■

Problem Set
Problem 3.

TRUE or FALSE?

1. Heap-sort is stable.
2. For keys that are comparable, it is possible to have heap operations ExtractMin and BuildMinHeap that operate in time $\Theta(1)$ and $\Theta(n)$ respectively.
3. Heap-sort sorts in-place.
4. The running time of BuildMinHeap is $\Theta(n)$.
5. HeapSort is in fact SelectionSort; we first build a Heap of n keys in linear time, and then select the largest, second largest, etc in additional $O(n \lg n)$ time.

Proof. 1. FALSE, 2. FALSE, 3. TRUE, 4. TRUE, 5. TRUE. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 4.

In a standard implementation of a heap, heap elements have arbitrary key values. Let us deal with a specific case where key values belong to a restricted range $0, \dots, d - 1$, where d is a constant, i.e. $d = O(1)$ (for example, think of d being 10 if it helps). We want to design a MinHeap whose Insert(x) and ExtractMin() operations take constant worst-case running time $O(1)$ on a heap that contains n keys. Design such a heap, and show how Insert and ExtractMin can be implemented to achieve the required worst-case time bound of $O(1)$.

Note: x is an element (object) that has attributes (data members) $x.info$ and $x.k$. The former contain information about x . The priority of the element is $x.k$. Thus it is possible for two $x \neq y$ that have $x.k$ equal to $y.k$.

Proof. The key is that d is constant. A naive approach is to have a heap of d priorities and in each one there is an attached linked list of elements with the same priority. If you try to insert a key you might have problems or make the code unreasonably complicated. The easier approach is to create a array A of linked lists. (Think of adjacency list.) In linked list $A[i]$ all elements with priority i are inserted (at the end of the list). Insert is quite easy. ExtractMin starts checking $A[0] \dots A[d - 1]$. If a list is not empty the first element is removed from the linked list and returned. Because the length of A is d and d is constant all ops are $\Theta(1)$ time! Fill in the details! ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's premises.

Problem Set

Problem 5.

Implement a `HeapIncrease(A,i,k)` for a MaxHeap that increases the priority of $old = A[i]$ from the current $A[i]$ to a new $A[i] = k$ where k is larger than the previous $A[i]$ value (i.e. the old value).

Proof. Idea: Change the value at node i as appropriately in step 2. If the value is indeed changed, propagate this value to the root of the tree as in Heap-Insert to maintain MAX-heap property for nodes located higher in the tree (as new value can only be larger than old one, heap-property is maintained for subtree-view rooted at i).

`Heap-Increase-Key(A,i,k)`

```
1. old=A[i]; j=i;
2. A[i]= max(A[i],k);
3. if old not equal A[i] //i.e. A[i] has been changed
4. do key=A[i];
5.   while j>1 and A[parent(j)] < key
6.     do A[j] = A[parent(j)];
7.       j = parent(j);
8.   A[j]=key;
```

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

■

Problem Set

Problem 6.

Give an $O(n \lg k)$ -time algorithm to merge k sorted sequences into one sorted sequence, where n is the total number of keys in all the input sequences.

Proof. The idea is simple. Take the largest key of each sequence and insert it into a MaxHeap with k keys. Such a Heap never grows beyond k in size, and has one key representative of each one of the k sequences (the largest key still there and not moved to the output) unless a sequence become empty because all of its keys moved to the output. For each key in the heap we record also the (index of the) sequence from which this key was extracted. For example if 25 was extracted from sequence 4, we add (25,4) into the heap with 25 being the key value and 4 the additional information maintained.

To fill the output sequence we perform n ExtractMax operations from the Heap, each ExtractMax followed by an Insert of the maximum key of the remaining sequence to which the Extracted key belonged. This way the size of the heap is maintained at k ; it can only grow below k if one of the sequences becomes empty (i.e. all its key moved to the output already). In our example if 25 of sequence 4 was the key that ExtractMax returns, the next largest key of sequence 4 will be added to the k -key heap from the input sequence 4. If that sequence has no more keys, then we leave the heap deficient of one key. The TOTAL number of ExtractMax operations is going to be n , the number of keys in the output, and each such operation involves a heap of no more than k keys i.e. requires $O(\lg k)$ for an Extract. The subsequent Insert to an Extract also takes that much time. Finding the next largest key of a sequence is an $O(1)$ time operation given that the sequences are sorted. Details follow.

Step 1. Let $j = n$ be the total length of the sorted sequences. Let the tail (largest key) of sorted sequence i be denoted by t_i , $i = 1, \dots, k$. Take the current k tails t_i , remove each one from corresponding sequence, form a heap based on the key values of the k tails in $O(k)$ time, and then update the t_i to point to the second largest key of the sorted sequences (as largest keys are in the heap and not in the sequence any more).

Step 2. ExtractMax the largest key M of the heap in $O(\lg k)$ and place it in the j -th position of the output sequence. Decrement j by 1. If M came off the l -th sorted sequence (i.e. it was before step 1 $M = t_l$) then remove the current tail of the l -th sequence and insert it into the $k - 1$ key heap in $O(\lg k)$, and update t_l appropriately.

Step 3. Repeat Step 2 until $j = 0$.

Step 1 takes $O(k)$ time. In step 2, every time M is removed, it is larger than any of the remaining heap or sorted sequence keys. Time for step 2 is $O(\lg k)$ for ExtractMax M or for HeapInsert of the next largest key of M 's sequence. As there are a maximum of n keys that are HeapInsert-ed or ExtractMax-ed in/from the heap the total running is $O(n \lg k)$. ■

Problem Set Problem 7.

d-HEAPS. Analysis of d -ary heaps. (Assume MAXheaps in the remainder).

A d -ary heap is like a binary heap but non-leaf nodes have d children rather than 2.

- How would you represent a d -ary heap in an array?
- What is the height of a d -ary heap of n elements in terms of n and d ?
- Give an efficient algorithm that extracts the maximum element from a d -ary max-heap. Analyze its running time in terms of n and d .
- Implement **Insert** for d -ary heaps and analyze its running time.
- Given an efficient implementation of an operation **Increase(A, i, value)** which first sets $A[i] = \max\{A[i], \text{value}\}$ and then properly updates the d -ary max-heap. Analyze the worst-case running time of **Increase**.

Proof. (a) A d -ary heap is to be represented by an one dimensional array A . $A[1]$ stores the root and $A[2], \dots, A[d+1]$ store the d children of the root. The j -th child of node indexed i is $d(i-1) + j + 1$ where $1 \leq j \leq d$. Therefore the children of i can have potential indexes $d(i-1) + 2$ to $di + 1$.

If the indexing starts from 0 i.e. the root is at $A[0]$, then the children of i have indices starting with $di + 1$ through $di + d$. The parent of node i is $\text{floor}((i-2)/d + 1)$, in other words $\text{floor}((i+d-2)/d)$ if the indexing starts from 1. If the indexing starts from zero, it is $\text{floor}((i-1)/d)$. A binary heap is a special case for $d = 2$.

(b) A heap of height h has $n = 1 + d + d^2 + \dots + d^{h-1} + m = (d^h - 1)/(d - 1) + m$ elements, where $1 \leq m \leq d^h$. Therefore its height is $O(\lg n / \lg d)$.

(c) Extract-Max works the same way as Heap-Extract-Max for binary heaps except that **DownHeap(A, i)** on d -ary heaps works a little bit differently. It assumes that the d subtrees of node indexed i have the heap property and the maximum is taken over the value stored at i and the values stored in all d children of i . The maximum operation takes time $O(d)$ now, and Extract-Max may call **DownHeap** $O(h)$ times. By way of part (b.b) $O(h) = O(\lg n / \lg d)$ and therefore the time for **DownHeap** will be $O(d \lg n / \lg d)$.

(d) Insert for binary heaps works fine for d -ary heaps as well (remember to use the new definition of a parent though). An upward movement is independent of the number of children of a parent. Therefore its running time is $O(h) = O(\lg n / \lg d)$.

Problem Set Continued

(e) This is similar to insertion. If the value k is at most the value of $A[i]$ nothing changes, otherwise the value of $A[i]$ is changed and propagated upwards just as we did in Heap-Insert. The observation is that since the value k is propagated upwards, we only need to change the value of the final destination only, just as we did in Heap-Insert. The following program demonstrates this. Note that $\text{parent}(i)$ follows the definition of part (b.a) not that of a binary heap.

Heap-Increase-Key (A, i, k)

1. If $A[i] \geq k$ then return ;
2. while $i > 1$ and $A[\text{parent}(i)] < k$
3. do $A[i] = A[\text{parent}(i)]$;
4. $i = \text{parent}(i)$;
5. $A[i] = k$;

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page



Problem Set Problem 8.

The following two methods are proposed to find M_k , the k -th smallest of n keys stored in array $A[0..n-1]$.

a) Sort A and then output the k -th element of the sorted A . This is M_k .

b) Build a MIN-heap out of the n elements of A , and then perform k EXTRACT-MIN operations. The last (i.e. k -th) EXTRACT-MIN operation returns M_k .

Which of the two algorithms is asymptotically faster? Explain, i.e. provide details of efficient implementation of the various steps, and analyze their worst-case running time using asymptotic notation. Use case analysis, if necessary.

Proof. The two implementations are shown in the figure.

a. FindMkOne ($A[1..n], n$)

1. HeapSort(A);

2. return($A[k]$);

b. FindMkTwo ($A[1..n], n$)

1. BuildMINheap(A, n);

2. for($i=1; i \leq k; i++$){

3. $z = \text{EXTRACTMIN}(A)$;

4. }

5. return(z);

(a) FindMkOne requires $O(n \lg n)$ worst-case running time for step 1 (HeapSort can be replaced by MergeSort) and step 2 requires constant time. The total is $O(n \lg n)$.

(b) FindMkTwo requires $O(n)$ worst-case running time for the build-heap operation of step 1, and $O(1)$ time for step 5. The loop involves k EXTRACT-MIN operations. Each one takes $O(\lg n)$ time for a total of $O(k \lg n)$. Note that the z returned in step 5, is the result of the k -th EXTRACT-MIN operation, the k -th smallest of the n keys of the input. Total running time is $O(n + k \lg n)$.

Let us asymptotically compare the performance of the two algorithms for various values of k . To make thing less confusing consider algorithm (a) has performance $n \lg n$ and (b) $n + k \lg n$.

Test Case 1. Let us start with small values of k . Let $k = \Theta(1)$, i.e. k is constant. The running time of (a) is $n \lg n$ and (b) is $n + 1 \lg n = \Theta(n)$. The latter algorithm is asymptotically faster.

How much can we increase k beyond a constant and still get the same performance? For what values of k is the term $k \lg n$ comparable to n ? This is for $k \lg n = O(n)$, i.e. $k = O(n/\lg n)$. For such values (b) is $\Theta(n)$ and (a) is $n \lg n$ and (b) is asymptotically faster.

Test Case 2. Suppose that $k = \omega(n/\lg n)$. Algorithm (b) has performance $n + k \lg n = n + \omega(n/\lg n) \lg n$ which is $\omega(n + (n/\lg n) \lg n)$ i.e. it is $\omega(n)$.

Problem Set Continued

Test Case 3. However as long as $k = o(n)$, the term $n + k \lg n$ is still $n + o(n \lg n) = o(n \lg n)$, i.e. algorithm (b) is still asymptotically faster than algorithm (a).

Summary Case. A.

Cases 1, 2, and 3 can be unified into: if $k = o(n)$ then (b) is asymptotically faster than (a).

Summary Case B. If k is not $k = o(n)$, then $k = \Theta(n)$, and algorithm (b) has performance $n + k \lg n = O(n \lg n)$ and both algorithms have the same asymptotic performance.

Conclusion: For large k such as $k = \Theta(n)$ both have the same asymptotic performance. For any other value of k i.e. $k = o(n)$ (b) is asymptotically faster than (a). ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 9.

You are given n arbitrary (but distinct) keys. We are interested in finding the $\sqrt{n}$ keys that are the smallest among the n -keys. Given an asymptotically optimal algorithm for returning/printing those keys in sorted (i.e. increasing) order.

Proof. Generalization of a prior Problem. Think!



Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set Problem 10.

Compress the text below (alphabetic characters only) using Huffman coding. What is the total number of bits used for the compressed text (ignore table storage requirements)?

ABRACADABRAZABRAZE BRA

Proof.

ABRACADABRAZABRAZE BRA

-x.-, -' -x.-/-x.-/;x.-

A: 8, B: 4, C: 1 D: 1, E: 1, R:4 Z: 2

	Frequency	PrefixCode	#of bits in code	Total # bit in 'compressed' text
	21			
/ \	8	A: 1	1	8
13 \	4	B: 011	3	12
/ \	4	R: 010	3	12
/ \	2	Z: 0011	4	8
/ \	1	E: 0010	4	4
/ \	1	D: 0001	4	4
/ \	1	C: 0000	4	4
2 3 8 \				----
/ \ / \ / \ \				52 <<<<<Answer
C D E Z R B A				Original Text (ASCII) Size : 21B or '168bit'
1 1 1 2 4 4 8				Compressed Text Size : ceil(52/8)=7B or ' 52bit'
				compression factor 21/7=3 168/52=3.x

Problem Set
Problem 11.

Using Huffman coding, how many bits do you need to compress this text of characters (count only bits of character representation)?

ILOVEALGORITHMS

Proof. Be careful! Love (of algorithms) hurts and takes time and practice! I just couldn't resist! ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 12.

How can you realize a data structure that supports ExtractMax, ExtractMin and Insert all three of them in $O(\lg n)$ time using structures already defined and used in class? Use as little space as possible.

Proof. An outline by skipping details. Use a Binary MinHeap and MaxHeap with two auxiliary arrays ToMax and ToMin. For a MinHeap[i] element, ToMax[i] shows its position (index) into the MaxHeap. Likewise for MaxHeap[i] element.

When an element is inserted it is inserted into the MinHeap first; in DownHeap when two elements are swapped say MinHeap[i] and MinHeap[j] to maintain the heap property, then make sure that the corresponding ToMin entries get swapped i.e. ToMin[ToMax[i]] and ToMin[ToMax[j]]. Then it is inserted into the MaxHeap similarly.

When Extraction is to be performed, say ExtractMax, the key is HeapExtracted from the MaxHeap first. Any DownHeap operations should update ToMax as well. When the key is erased from the MaxHeap, its ToMin[0] entry of the MinHeap must be 'deleted' and 'not extracted'. Naive 'deletion' would create a gap: replace the gap with the last element as in an Extract operation. From that point on it works either as a DownHeap if the replaced value is smaller than the value replacing it, or upwards as an Insert operation otherwise. See example below and try to "delete" the 10 or the 25. The ExtractMin operation is analogous. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web page