Elementary Data Structures

CS 114/116 and CS 505 material

Chapters 2.1-2.3, 4.2.2 of GT

Disclaimer: These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2016; distribution outside this group of students is NOT allowed.
Dynamic Sets
Overview

Sets as well as operations on sets are of paramount importance in computer science. Although in a mathematical context a set is a static concept, in computer science a set becomes a dynamic entity whose size can change during the course of the execution of a program/algorithm.

Such sets are called dynamic to distinguish them from static ones.

For a dynamic set we need to introduce the operations that operate on these sets and describe algorithms that implement these operations.

Such operations can be dynamic or static. Dynamic operations are operations such as Insert, Delete, ExtractMax or DeleteMax, ExtractMin or DeleteMin, that modify the set and are also known as modifying operations. Static operations are operations such as Search, Min or FindMin, Max or FindMax, Predecessor and Successor that do not modify the set but just probe it. They are also known as probing operations.

Issues related to the efficiency of these operations need to be dealt with as well. Foremost and related to such issues is the problem of the representation and storage of a set in the memory of a computer.

An abstract data type that is a dictionary supports operations like insertion of a word, deletion of a word and search of a word in the dictionary. It can be used to maintain a set of words (in the dictionary).

In the following discussion \( U \) is a universe of elements, \( S \subseteq U \) denotes a set and \( x \in U \) is an element of \( U \). Each \( x \) has associated with it a field \textit{key} with value \( k \). Other information may also be associated with \( x \). Without loss of generality we shall ignore this additional information as for the operations that we shall describe for a set \( S \), this additional information does not affect the description of the operations.
Dynamic Sets
Operations

**Search(S,k)**
Given a set $S$ and a key value $k$, decide whether there exists an element $x$ of $S$ whose key is $k$. If there exists such an $x$ return a pointer to $x$, otherwise return NIL.

**Insert(S,x)**
Given a set $S$, $x$ is inserted into $S$.

**Delete(S,x)**
Given a set $S$, and a pointer to an element $x$ of $S$, element $x$ is deleted.

**Min(S)**
Given a set $S$ find the element of $x$ with the minimum key value and return the value (or the element with the min value).

**Max(S)**
Given a set $S$ find the element of $x$ with the maximum key value and return the value (or the element with the max value).

**Successor(S,x)**
Given $x$ and $S$ the element of $S$ is returned with value the next larger to the value $k$ of $x$ or nil.

**Predecessor(S,x)**
Given $x$ and $S$ the element of $S$ is returned with value the next smaller to the value $k$ of $x$ or nil.
Stacks and Queues
A short review

Stacks and queues are dynamic sets where the element that is being deleted by a DELETE operation is prespecified. In a stack this element is the most recently inserted into the stack. In a queue it is the element least recently inserted. For this reason the two structures are also known as LIFO (last-in first-out) and FIFO.

For a stack, the Insert and Delete operations are more often called Push and Pop.

An implementation of a stack in terms of an array is immediate. The leftmost end of the array is the bottom of the stack, the stack grow to the right and the top of the stack is pointed by a pointer top. A stack with no elements is an empty stack. If an operation tries to reach an element past the left end an underflow has occurred. If it tries to reach an element past the right end an overflow is flagged.

For a queue, the Insert and Delete operations are more often called Enqueue and Dequeue. A queue has a head and a tail. A dequeue operation removes the element of the head, and an enqueue operation adds an element at the tail of the queue.

An implementation of a queue in terms of an array is easy if we use a floating pointer to denote the head and the tail of the queue. Initially head and tail point to the beginning of the queue. If head=tail+1, the queue is full. When they are equal the queue is empty. Head points to the first element and tail next to the last element (i.e. the first available position). When tail or head reaches n (size of array) the value is rolled over to 1, that is, we visualize the array as a circular one.
Stack and Queue Operations
A short review

EmptyStack(S) PUSH(S, x) POP(S)
1. if top(S)==0 top(S)=top(S)+1; if EmptyStack(S)
2. return(TRUE); S[top(S)] = x; return(UNDERFLOW);
3. else else
4. return(FALSE); top(S)=top(S)-1;
5. return(S(top(S)+1));

1. Enqueue(Q, x) Dequeue(Q)
2. Q[tail(Q)]=x; x=Q[head(Q)];
3. if tail(Q) == length(Q) if head(Q)==length(Q)
4. tail(Q)=1; head(Q)=1;
5. else else
6. tail(Q)++; head(Q)++;
Linked List
Overview

An array is a random access data structure. We can access the $i$-th element of the array in unit time. Whether we access the 1st, $i$-th or last element, the time is the same. Random Access means the same as the term Random Access Memory (RAM).

A linked-list is a data structure whose objects are arranged in a linear order. Random access to the $i$-th element of the list is not supported (as it was, in the case of an array): it takes $i$ steps for access the $i$-th element of a linked list. A pointer structure is used however to represent a linked list.

A doubly linked list is a list whose objects are arranged in a linear and reverse linear order. Pointers prev and next point to the following element in the list in each of these two orders (forward and backward). A doubly linked list allows for constant time deletion of an element pointed by a pointer/reference.

In a singly linked list only one pointer is available (next). If the list is sorted according to the value of the keys of its elements, the linear order of the list is the linear order of its keys.

In a circular list next of last element (tail) is the first (head).

Important operations in a linked list: Search, Insert, Delete.

What is a Sentinel? We may avoid the use of null pointers by making such elements (pointing to null) to point to a sentinel element (a default-non-valid-list-element) that points to the head of the list.
Implementing Pointers
Overview

If we have say a doubly-linked list with two pointers next, prev, and key, we can represent such a list by three arrays one for each of next, prev, and key. A pointer value then becomes an index value of the array.

Sometimes it makes sense to represent a list in a single array. Then, an element of the list will occupy space \( m \) which is large enough to store prev, next, and key (for the first two an array index is stored for each prev, next). A pointer points to the first position of size \( m \) allocated to the pointed linked-list element.

As a linked-list is a dynamic data structure in both representations elements are added to and deleted from the list. If the arrays/array are/is of size \( n \) a naive approach of implementing linked-list operations on these structures may cause problems after the first \( n \) operations.

We need to somehow record the freed positions (after a delete operation) and reallocate these positions whenever an Insert is issued. The garbage-collector is responsible for such allocation/deallocation operations.

To allow this we keep in the free list all available memory (ie it is a linked list). Initially the whole memory belongs to the free list. Whenever a portion of the array is deleted it is added to the top of the free list and whenever an element needs to be allocated it is allocated from the top (ie it is a stack). **WHY???.** Every element is either in the list \( L \) or in the free store.

This is one reason why it is advised that a malloc(A), malloc(B) in the C programming language be followed by a free(B) and free(A). **Why is it so?**

The methods we used to represent lists extend to representing any homogeneous data structure including trees.

We can represent a binary tree by using three fields, left and right to point to the children of the node/element of the tree and \( p \) to point to its parent. An element with both left and right NIL is a leaf. An element with \( p \) NIL is the root pointed to by root(T).

Suppose that a tree node has unlimited degree (number of children). Then left(\( x \)) points to the left child \( y_1 \) of \( x \) and right (\( y_1 \)) points to its sibling \( y_2 \) (second child of \( x \)) and so one until the last one \( y_k \) points to nil (no more siblings).
Graphs
Introduction

A directed graph/digraph $G$ is a pair $G = (V, E)$ where $V$ is a finite set and $E$ is a binary relation on $V$ (i.e. a subset of $V \times V$). $V$ is called the vertex set and $E$ the edge set of $G$. The elements of both sets are called respectively the vertices and edges of $G$. In a directed graph $(u, v)$ represents an edge leaving $u$ and entering $v$. In a digraph self-loops are possible. Multiple edges are not allowed. In the directed case edge $(u, v)$ is incident from $u$ or leaves vertex $u$ and is also, incident to $v$ or enters $v$. The endpoints of $(u, v)$ are $u$ and $v$.

For directed graphs, we sometimes use the notation nodes and arcs to denote the vertices and edges respectively. Under such a notation we may use $G = (N, A)$ instead of $G = (V, E)$.

An undirected graph $G$ is a pair $G = (V, E)$ where $V$ is a finite set and $E$ is a set of unordered pairs of vertices. We represent an edge as $(u, v)$ even if we really mean $\{u, v\}$. In an undirected graph self-loops are not allowed. Multiple edges are not allowed. For the undirected case $(u, v)$ is incident on $u$ and $v$. Two vertices connected by an edge are called adjacent, i.e. $u$ is adjacent to $v$. For an undirected graph $(u, v)$ and $(v, u)$ are the same edge. The endpoints of $(u, v)$ are $u$ and $v$.

By convention we use the same notation to represent an edge for the directed and undirected case, that is $(u, v)$ is an edge from $u$ to $v$ in the directed case, whereas in the undirected case it represents an edge between $u$ and $v$.

If $S$ is a set of vertices an edge is incident to $S$ if exactly one of its endpoints is in $S$.

A graph $G = (V, E)$ is bipartite if there is a set $S \subseteq V$ such that every edge of $G$ is incident to $S$. 
Graphs
Notation

In the directed case the **out-degree** of a vertex $u$ is the number of edges incident from $u$.
In the directed case the **in-degree** of a vertex $v$ is the number of edges incident to $v$.
The **degree** of a vertex in a directed graphs is the sum of its out-degree and in-degree.

In the undirected case the **degree** of a vertex $u$ is the number of edges incident on $u$.

A **path** of length $k$ from $u$ to $v$ is a sequence $\langle u_0, u_1, \ldots, u_k \rangle$ of vertices such that $u_0 = u$ to $u_k = v$ and $(u_i, u_{i+1}) \in E$, $0 \leq i < k$. $u_0 = u$ and $u_k = v$ are the endpoints of the path. The length of the path is the number of edges in it. We also say that $u_i$ is **reachable** from $u$. A path is **simple** if the are no duplicate vertices (ie all vertices in the path are distinct). A subpath of a path is a contiguous subsequence of its (path’s) vertices.

A **cycle** in a directed graph is a path with $k \geq 0$ whose endpoints coincide ie $u = v$ and has at least one edge. A cycle is **simple** if all $u_1, \ldots u_k$ are distinct. A self-loop is a cycle of length 1.

A cycle in an undirected graph is defined identically (self-loops are not allowed in undirected graphs) with the addition that no edge is repeated.

A graph with no cycles is called **acyclic**.

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. $G_1$ is a subgraph of $G_2$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$. 
An undirected graph is **connected** if every vertex $u$ is reachable from every other vertex $v$. The **connected components** of a graph are the equivalence classes under the “is reachable from” relation or in other words they are the maximal connected subgraphs of $G$.

An undirected graph is connected if number of connected components is 1.

A directed graph is **strongly connected** if every two vertices are reachable from each other. The **strongly connected components** of a graph are the equivalence classes under the “are mutually reachable” relation.

Two graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there exists a bijection (surjective and injective function) $f$ from $V$ to $V'$ such that $(u, v) \in E$ is mapped to $(f(u), f(v)) \in E'$.

A **complete graph** on $n$ vertices is a graph such that every pair of vertices is adjacent. It is denoted by $K_n$.

A **forest** is an acyclic undirected graph.

A **(free) tree** is a forest that is also connected.
A **free tree** is an connected acyclic undirected graph. A connected forest is thus a tree. We often omit *(free)* from *(free)* tree.

**Theorem.** Let $G$ be an undirected graph. The following statements are equivalent

1. $G$ is a free tree.
2. Any two vertices in $G$ are connected by a unique single path.
3. $G$ is connected but if any edge is removed from $E$ the resulting graph is disconnected.
4. $G$ is connected and $|E| = |V| - 1$.
5. $G$ is acyclic and $|E| = |V| - 1$.
6. $G$ is acyclic but if any edge is added to $E$ the resulting graph contains a cycle.
Rooted Trees
Introduction

A **rooted tree** is a free tree in which one of the vertices is distinguished from the others and is called the **root**. The root of a tree $T$ will be represented sometimes by $r$. Vertices of a tree are also called **nodes**.

Let $P$ be a path from $r$ to some vertex of $T$. Then any node $y$ of the unique path from $r$ to $x$ is called an **ancestor** of $x$. $x$ is a **descendant** of $y$ ($r$ and itself). If $x \neq y$ then the $x$ is a proper descendant of $y$ and $y$ is a proper ancestor of $x$. $x$ is an ancestor/descendant of $x$. The subtree rooted at $x$ is the tree induced by descendants of $x$ rooted at $x$.

If in the path the last edge from $r$ to $x$ is $(z, x)$ $z$ is called **parent** of $x$, $x$ is called **child** of $z$. If $x$ and $u$ are both children of $z$ they are called **siblings**.

A node with no children is called a **leaf** or **external node**. A non-leaf is called an **internal node**.

The number of children of a node defines its **degree** (note the difference between this definition and the degree of a vertex in a graph).

The length of the path from $r$ to $x$ is the depth of $x$. A root $r$ has depth $d(r) = 0$. If $v$ is a tree node the $d(v) = d(p(v)) + 1$ where $p(v)$ is parent of $v$. The largest depth of any node is the height of the tree $T$. The height of a leaf $v$ is $h(v) = 0$. For a non-leaf node $u h(u) = \{ \max h(w), \text{ where } w \text{ is a child of } u \}, +1$.

An **ordered** tree is a tree in which the children of a node are ordered (ie we can talk of the first, second etc).
A **Binary Tree** is a tree that either

- contains no nodes, or

- it consists of three disjoint sets of nodes: a **root** node, a binary tree called **left-subtree** and a binary tree called **right-subtree**.

The root of the left subtree is called the left child of **root** and that of right subtree the right child.

**Difference between binary trees and ordered trees.**

If a node has only one child it matters whether it is a left or a right child (as opposed to ordered trees, where it doesn’t matter).

A **full-binary tree** is a tree where each node has degree 2 or 0 (leaf).

ATTENTION !

A **complete binary tree** is a tree where each leaf has the same depth and all internal nodes have degree 2.

The previous definition differs from the one in the textbook (page 99) given in the context of binary heaps. We shall call the tree that has the property of page 99 a **heap tree** or **complete (binary) heap tree**.

END OF ATTENTION.

Number of vertices in such a tree of height \( h \) is

\[
1 + 2 + 2^2 + \ldots + 2^h = 2^{h+1} - 1.
\]

and the number of internal nodes is \( 2^h - 1 \).

Example. Suppose that internal node of depth \( i \) are \( d_i \) and internal nodes of depth \( i + 1 \) are \( d_{i+1} \). In a full binary tree how many leaves there are of depth \( i + 1 \)? Answer is \( 2d_i - d_{i+1} \).

Example. If the height of the tree is \( h \) how many leaves of depth \( h \) there are?

Answer. There are no internal nodes of depth \( h \) (otherwise we would have had nodes in depth \( h + 1 > \) height of tree. Thus internal nodes of depth \( h - 1 \) is by our notation \( d_{h-1} \). Therefore number of leaves is \( 2d_{h-1} \) at depth \( h \).
Graph representations

Introduction

For a graph $G = (V, E)$ let $n = |V|$ and $m = |E|$.

A graph $G$ is **dense** if $m \gg n$, and **sparse** otherwise. In some cases the former condition is restated as $m = \Theta(n^2)$ and the latter condition $m = o(n^2)$.

There are two major ways one can use to represent a graph

1. **Adjacency matrix** representation. A graph $G = (V, E)$ on $n$ vertices and $m$ edges is represented by an $n \times n$ two-dimensional array $A$. Element $A(i, j)$ is 1 if there is an edge $(i, j)$ in the graph, otherwise it is 0. For undirected graphs, $A$ will be symmetric since for every edge $(i, j)$ both $A(i, j)$, $A(j, i)$ are set to 1. In general, $m$ entries in the directed case and $2m$ entries in the undirected case are 1, whereas the remaining ones are 0. If the edges of the graph have weights, then instead of storing an 1 we store in $A$ the weight of the edge. Total space requirements are $O(|V|^2)$.

2. **Adjacency list** representation. The graph is represented by $n$ linked lists, one for each vertex of $G$. The linked list for vertex $i$ contains all vertices that are neighbors of $i$, i.e., it contains all $j$ such that $(i, j)$ is an edge. For an undirected graph, $j$ is in the linked list of $i$ and $i$ is in the list of $j$. Total space requirements are $O(|E|)$. The problem with this representation is that we cannot answer quickly questions of the form "Is $j$ adjacent to $i"?"

Just for information purposes, the **incidence matrix** of a directed graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = [b_{ij}]$ such that $b_{ij} = -1$ if edge $j$ leaves vertex $i$, $b_{ij} = +1$ if edge $j$ enters vertex $i$, $b_{ij} = 0$ otherwise.
Introduction

A **tree traversal** is a process of visiting each of the vertices of a rooted tree exactly once. For rooted trees preorder and postorder tree traversals are defined. For the specific case of a **binary** tree three such traversal are quite known: **inorder**, **postorder**, **preorder**. In preorder, parents are visited before their children; in postorder after their children; for binary trees, left children are visited before right children. Inorder traversal has a meaning only for binary trees: left subtree of a parent is visited before the parent followed by right subtree. As a rooted tree may or may not be ordered, inorder is not well-defined/meaningful.

```plaintext
pre-visit(u) :: {pre[u]=pre++; print(<u,pre[u]>);} ;
in-visit(u) :: {in[u]=in++; print(<u,in[u]>);} ;
post-visit(u) :: {post[u]=post++; print(<u,post[u]>);} ;
```

- **BT-Inorder(u)**
  1. if u != NULL {
  2. BT-Inorder (left(u));
  3. in-visit(u) ;
  4. BT-Inorder (right(u));
  }

- **BT-Preorder(u)**
  1. if u != NULL {
  2. pre-visit(u);
  3. BT-Preorder(left(u));
  4. BT-Preorder(right(u));
  5. post-visit (u);

- **BT-Postorder(u)**
  1. if u != NULL {
  2. pre-visit(u);
  3. BT-Postorder(left(u));
  4. BT-Postorder(right(u));
  5. post-visit (u);

- **BT-Perform-All-Three(u)**
  1. pre-visit(u);
  2. if left(u) != NULL
  3. BT-Perform-All-Three (left(u));
  4. in-visit(u);
  5. if right(u) != NULL
  6. BT-Perform-All-Three (right(u));
  7. post-visit (u);

- **Euler-tour(u)**
  1. left-visit(u); // similar to pre-visit
  2. if left(u)!=NULL
  3. Euler-tour(left(u));
  4. down-visit(u); // similar to in-visit
  5. if right(u) != NULL
  6. Euler-tour(right(u));
  7. right-visit(u); // similar to post-visit
Another useful traversal is the **breadth-first order** (BFO) traversal obtained as follows.

\[
\text{RT-BFO(} \text{root(RT))} \quad \text{// RT is a rooted (not necessarily binary) tree} \\
1. \text{visit(} \text{root(RT))}; \quad \text{// visit[root(RT)]=i++; print(visit[root(RT)])}; \\
2. \text{repeat until all vertices are visited} \\
3. \text{visit an unvisited child of the LEAST RECENTLY VISITED} \\
\quad \text{vertex with an unvisited child} \\
\text{END_BFO}
\]

If **LEAST RECENTLY** is replaced by **MOST RECENTLY**, a **depth-first order** (DFO) traversal is obtained.

We can extend these two traversals from rooted trees to graphs. If \(G\) is a graph and \(s\) is an arbitrary starting vertex, we use this vertex as the “root” of the traversal by visiting \(s\) first and then repeating the following step until there is no unexamined edge \((u, v)\) such that vertex \(u\) has been visited.

**Search Step.** Select an unexamined edge \((u, v)\) such that \(u\) has been visited and examine this edge, visiting vertex \(v\) if \(v\) has not been visited yet.

If \(u\) is the MOST RECENTLY VISITED vertex we get a DEPTH-FIRST SEARCH. In BREADTH-FIRST SEARCH \(v\) is the least recently visited vertex.
A **disjoint-set data structure** maintains a collection of $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets consisting of $n$ elements. Each set is identified by a **representative** (i.e., leader) which is some arbitrary member of that set. In many applications it doesn’t matter which member becomes the representative of the set. We are satisfied that our queries return the same result as long as the sets are not modified. There are some cases where a representative may possess some additional property (i.e., it is the smallest element of a set with some order) but we are not interested in it right now.

For an object $x$ we wish to support in such a data structure the following operations.

1. **MakeSet($x$)** creates a new set whose only member is pointed to by $x$. Since sets are disjoint we assume that $x$ is not already in a set.

2. **Union($x, y$)** unites the dynamic sets that contain $x$ and $y$ say $S_x$ and $S_y$ into a new set that is the union of these two sets. The two sets are disjoint prior to the operation. The representative of $S_x \cup S_y$ is some member of that set (not necessarily $x$ or $y$). The two old sets $S_x$ and $S_y$ are destroyed after the creation of $S_x \cup S_y$.

3. **Find($x$)** returns a pointer to the representative of the unique set containing $x$ (parameter $x$ is a pointer to $x$).

We are interested in operations where $n$ is the number of elements and thus the maximum number of MakeSet operations and $m$ is the total number of all three operations. Note that after $n - 1$ Union operations only one set may remain. Also, $m \geq n$. 
Disjoint-Sets
An application

An application of disjoint sets is in finding the connected components of an undirected graph $G$.

Connected-Components($G$)
1. for each vertex $u$ in $V$
2. do MakeSet($u$)
3. for each edge $(u,v)$ in $E$
4. do if Find($u$) $\neq$ Find($v$)
5. then Union($u,v$)

Same-Components($u,v$)
1. if Find($u$) $==$ Find($v$)
2. then return(TRUE)
3. else return(FALSE)

What is the running time of the two algorithms?
case 1. Adjacency list representation of the graph
case 2. Adjacency matrix representation of the graph.

What is the running time of MakeSet, Union, Find? In order to answer the question about the running time of Connected-Components and Same-Component we need to determine the running time of these three operations. This depends on the implementation of the data structure that supports disjoint sets.

Note 1. The number of MakeSet operations performed in line 2 of Connected-Components is $n$, the number of elements i.e. the number of vertices of the graph. The number of Find($u$) or Find($v$) operations depends on the number of edges $m$. it would be $2m$ or $4m$ depending on whether the graph is directed or undirected. For an undirected graph bother $(u,v)$ and $(v,u)$ is recorded as an edge so we double the work performed. The number of iteration of line 3 depends on the representation of the graph. It might be $O(n^2)$ or just $O(m)$.

Note 2. The running time of Same-Components is the running time of Find (or twice of it)!
Disjoint Set Representation
Linked-List

We represent each set by a linked list whose head points to the representative and tail to the last element of the set, and the remaining elements are the remaining elements of that set (as in an ordinary linked list). Each element has a pointer not only to the next element but also to the representative of the set containing it. MakeSet(x) runs in time $O(1)$ as it creates a list with one element. Find(x) also runs in time $O(1)$ as we return the pointer to x’s representative; the parameter to find is a pointer to $x$ and thus x’s representative can be accessed in the claimed time.

Union(x,y) is a bit more difficult to implement. For two lists we append x’s list onto the end of y’s list. We need to go through the elements of x’s list to update pointer representative. In the worst case we may spend $O(m^2)$ operations.

MakeSet(x1).... MakeSet(xn);
Union(x1,x2);Union(x2,x3);Union(x3,x4); etc

How do we decrease the running time

Solution. Append the smallest of the two lists onto the end of the largest one. Every time a list is appended the resulting list doubles in size (in terms of the smallest one). Consider a fixed element $x$. Every time its representative pointer is updated in a union operation $x$ is part of the smaller of two sets whose size is subsequently doubled. Therefore $x$’s representative pointer is updated at most $\lg n$ times as after so many updates $x$’s set would contain all set elements (unions have resulted in a single set). We have $n$ set members, each one is updated at most $\lg n$ times for a total update cost and also runtime for union operations of $O(n \lg n)$.

Collective running time. For $m$ operations with $n$ MakeSet total cost is thus $O(m + n \lg n)$ since all other operations are $O(1)$.

Amortized running time. This collective running time gives rise to the amortized running time of an operation. Divide the running time by $m$ and the answer is $O(1 + n \lg n/m)$. 