Basic Data Structures

IE 496 Lecture 11

Reading for This Lecture

- Horowitz and Sahni, Chapter 2

Basic Data Structures

What is a data structure?

- Data structures are schemes for organizing and storing sets.
- Data structures make it easy to perform certain set operations.
- Examples of set operations.
  - add
  - delete
  - find_min
  - delete_min
  - union

Choosing the right data structure

- Data structures consist of
  - a scheme for storing the set(s), and
  - algorithms for performing the desired operations
- Hence, each set operation has an associated complexity
- To choose a data structure, you should know
  - something about the elements of the set, and
  - what operations you will want to perform on the set.

Example: Lists

- A list is a finite sequence of elements drawn from a set.
- List operations
  - Create a list
  - Get the number of items
  - Get the value of item j
  - Set the value of item j
  - Add something to the list before item j
  - Delete item j from the list
- Lists have two basic implementation schemes.

A List Class

```
class list {
  private:
    // Here is the implementation-dependent code
    // that defines exactly how the list is stored.
  public:
    // Here are the operations to be implemented.
    // Create and destroy a list
    list();
    ~list();
    // Get the number of items in the list
    int getNumItems() const;
    // Get the value of item j
    bool getValue(const int j, int& value) const;
    // Get the value of item j
    bool setValue(const int j, const int value);
    // Add an item before item j
    bool addItem(const int j, const int value);
    // Delete item j
    bool delItem(const int j);
}
```

Implementing with Arrays

This source would be put in a file called list.h.

```
class list {
  private:
    // Here is the implementation-dependent code.
    // We’ll store the data in this array.
    int* array_;
    // Here is the size of the array.
    int size_;
    // Here is the number of items in the list.
    int numItems_;
  public:
    list();
    ~list();
    int getNumItems() const;
    bool getValue(const int j, int& value) const;
    bool setValue(const j, const int value);
    bool addItem(const int j, const int value);
    bool delItem(const int j);
}
```

Constructing and Destructing

This source would be put in a file called list.cpp.

```
#include "list.h"

list::list() :
    array_(new int[MAXSIZE]),
    size_(MAXSIZE),
    numItems_(0)
{}

list::~list() {
    delete array_;
    size_ = 0;
}
```

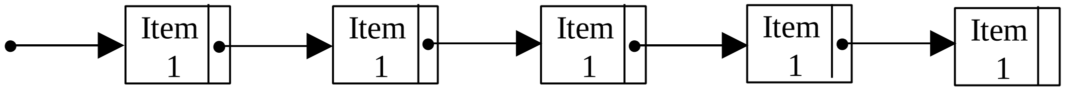
Implementing List Query Operations

```
int list::getNumItems() const {
    return numItems_;
}

const bool list::getItem(const int j, int& value) {
    if (j > 0 && j < size_){
        value = array_[j];
        return true;
    }else{
        return false;
    }
}
```

List Modification Operations

```
bool list::addItem(const int j, const int value){
    if (numItems_ == size_ || j < 0 || j > size_){
        return false;
    }else{
        for (int i = size_; i > j; i--)
            array_[i] = array_[i-1];
        array_[j] = value;
        size_++;
    }
}

bool list::delItem(const int j){
    if (j < 0 || j > size_ - 1){
        return false;
    }else{
        for (int i = j; i < size_ - 1; i++);
        array_[i] = array_[i+1];
        size_--;
    }
}
```

Linked Lists



	NAME	NEXT
0	-	1
1	Item 1	3
2	Item 2	0
3	Item 3	4
4	Item 4	2
5	Empty	0

Linked List Operations

INSERT

	NAME	NEXT
0	-	1
1	Item 1	3
2	Item 2	0
3	Item 3	5
4	Item 4	2
5	New Item	4

DELETE

	NAME	NEXT
0	-	1
1	Item 1	5
2	Item 2	0
3	Empty	0
4	Item 4	2
5	Item 5	4

Implementing with a Linked List

- For a linked list implementation, we replace the array with a linked list.
- To clients, the class would look exactly as before.
- Below is the definition of the linked list node class

```
class node {
  private:
    int value; // The value stored at the node
    node* next; // Pointer to the next node
  public:
    node();
    ~node();
    int getValue() const;
    int setValue(const int value);
}
```

Linked List Analysis

- `list()`
- `addItem()`
- `delItem()`
- `concatenate()`
- `split()`

Data structures in algorithms

- Typically, data structures are part of a larger algorithm.
- In order to choose a data structure, you should also know something about the algorithm.
- The data structure should be efficient for the operations that will be performed most often.
- The same algorithm can have different running times using different data structures.

Arrays vs. Linked Lists

- Linked lists
  - Efficient to add, delete, concatenate, split.
  - Don't have to know the number of data items in advance.
- Arrays
  - Less storage space.
  - Fewer memory allocations.
  - More efficient to locate i^{th} data item.

Using lists

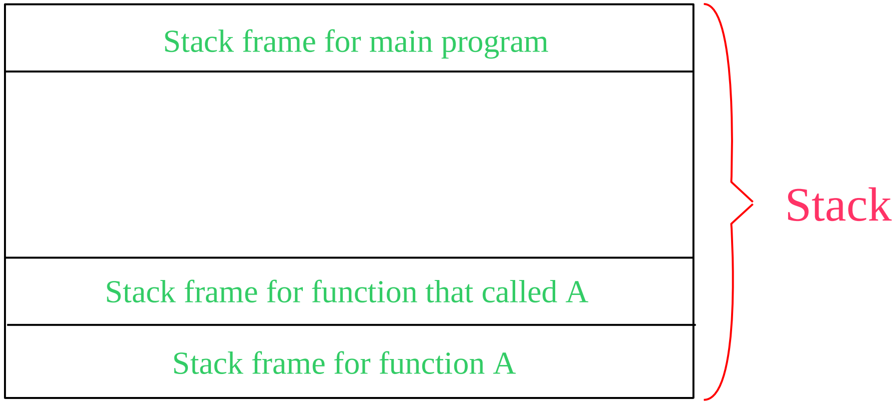
- Insertion sort
- Merge sort/quick sort
- Binary search
- Circular lists
- Doubly linked lists

Stacks

- A list data structure in which insertions and deletions are made at one end is called a *stack*.
- This is also known as a Last In First Out (LIFO) list.
- Insert and delete operations are often called *push* and *pop*.
- Stack Data Structures
  - Array
  - Linked list
- Stacks can be used to keep track of data in recursion (stack frames).

Stack Frames

- Local data for each function call is stored on the *stack*.
- Each function gets a *stack frame* to store data.
  - space for local variables.
  - pointers to the parameters the function was called with.
  - pointer to the instruction to return to in the calling function.
  - pointer to the localtion to store the return value.



Stack frame for main program

Stack frame for function that called A

Stack frame for function A

Stack

Queues

- A queue is a list in which insertions take place at one end and deletions at the other.
- Also known as First In First Out (FIFO) lists.
- Insert and delete operations are often called *enqueue* and *dequeue*.
- Queue data structures
  - Array
  - Circular array
  - Linked list

Graph Terminology

- Given a directed graph *G = (V, E)*, we define
  - a *path* is a sequence of edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$.
  - such a path is said to go *from vertex* 1 *to vertex* n.
  - A path is *simple* if no two edges on the path share a common endpoint, with the exception that we allow $v_1 = v_n$.
  - A simple path in which $v_1 = v_n$ is called a *cycle*.
  - A directed graph with no cycles is called a *directed acyclic graph*.
  - For vertex w, the number of edges (v, w) in G is called the *in-degree* of w.
  - Simlarly for *out-degree*.

Graph Data Structures

- Recall: Graph consists of
  - A set of *nodes* or *vertices* V.
  - A set of *edges* $E \subseteq V \times V$.
- Adjacency matrix
  - Efficient for determining whether a particular edge is present.
  - Requires $O(|V|^2)$ storage and initialization time.
- Adjacency lists
  - Usually the method of choice.
  - More efficient for sparse graphs.

Trees

- A (directed) tree is a directed acylic graph satisfying the following:
  - There is exactly one vertex called the root with in-degree 0.
  - Every other vertex has in-degree 1.
  - There is a path from the root node to every other node.
- Trees also have a natural recursive definition.
- Tree terminology
  - If $(u, v) \in E$, then u is called the *parent* of v and v is called the *child* of u.
  - If there is a path from u to v, then v is a *descendant* of u and u is an *ancestor* of v.

More Tree Terminology

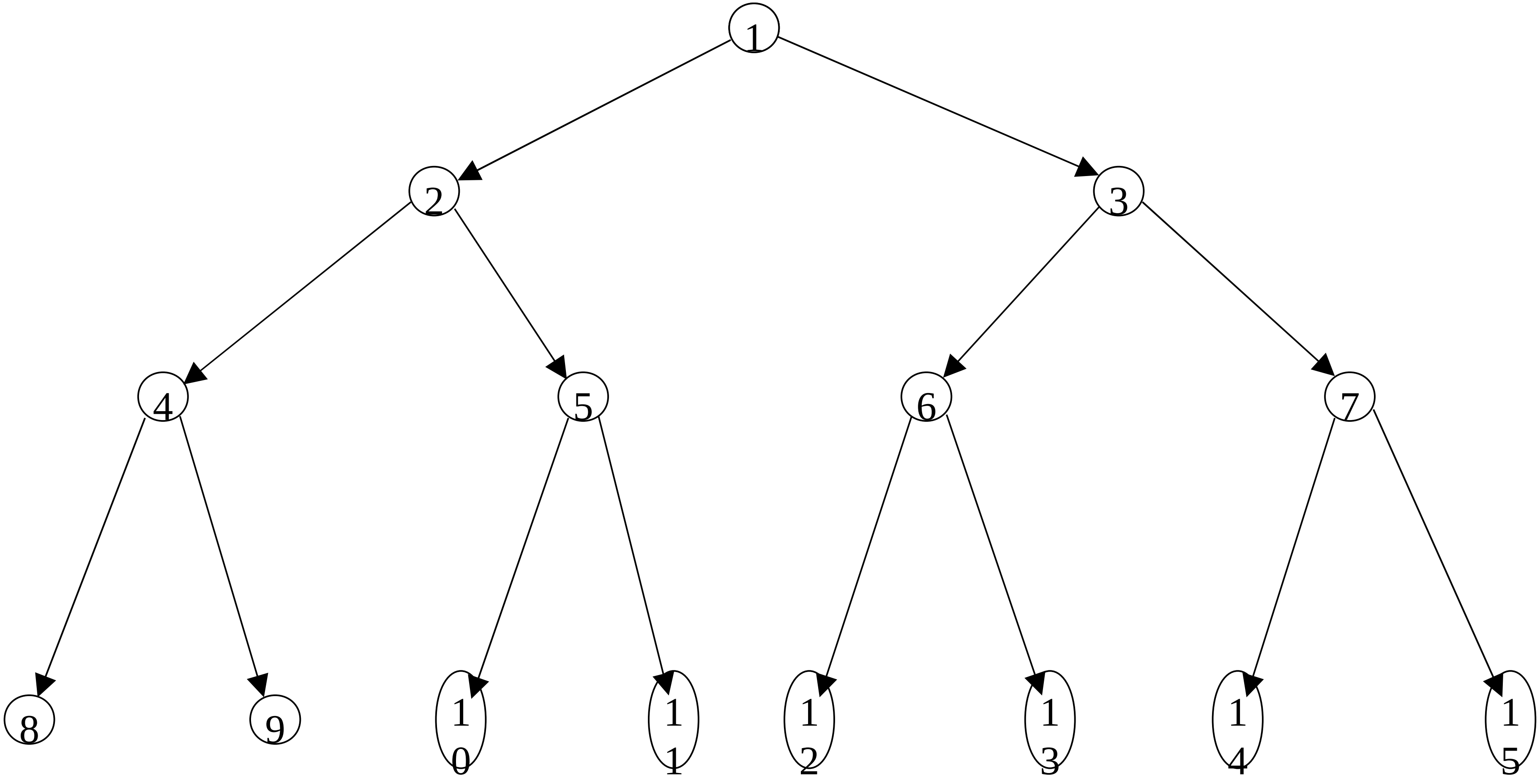
- A tree in which each node has out-degree 0, 1, or 2 is called a *binary tree*.
- A *balanced tree* is one in which all leaves are at levels k or k-1.
- In a binary tree, the two children are usually distinguished as the *left child* and the *right child*.
- The *depth* or *level* of a vertex v is the length of the (unique) path from the root to v.
- The depth of a tree is the maximum depth of any node.

Trees and data structures

- Trees are an element of many different data structures.
- Trees are naturally associated with recursive and divide and conquer type algorithms.
- Sample tree operations
  - `parent(), right(), left()`
  - `delete()`
  - `add()`
  - `link()`

Storing a binary tree

- Arrays
  - Parent of node i is stored in location $\lfloor i/2 \rfloor$.
  - Easy to go to a specific node.
  - Can use up lots of memory if unbalanced (2^l elements).
  - Not efficient for some tree operations.
- Pointers
  - Can be more memory efficient if unbalanced.
  - Easier tree operations in some cases.


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Traversing a Tree

- Many common algorithms involve traversing or searching a tree.
- Traversal schemes
  - preorder
  - postorder
  - depth-first
  - breadth-first