Lists, Stacks & Queues

Outline

- We assume we are familiar with basic data types (and their operations): arrays, 2D arrays, pointers, structs, etc.
- Refresh elementary linked lists (insertion, deletion, length, etc.)
- ADTs: Stacks, queues.
- Doubly linked lists and circular lists.
**Linked List Basics – Warm-up**

There are two basic mechanisms to work with lists (collection of similar items): (1) Arrays (2) Linked Lists

**Arrays:**
- **Advantages:** once an array is declared and set up, the [ ] operators makes random access extremely fast on hardware using fast address arithmetic.
- **Disadvantages:** The size needs be declared (can be done only once) – no dynamic allocation of space – wastage of space in most cases [programming efforts using realloc() in the heap area can minimize – inconvenient, complex]. Insertion/deletion of elements is time consuming [existing elements have to be moved around]; but can have excellent efficiency for storage and iteration., especially since modern memory systems are tuned for the access of contiguous areas of memory.

**Linked lists:** It can dynamically allocate/deallocate storage with minimal effort and insertion/deletion of elements is also fast – there are many other potential advantages to create new types of structures. Random access is typically slow [e.g. no [ ] operator.]

**Lesson:** Choose your data structure to suit your problem and application.

Read here for linked list basics; Read here for a refresher on C. Remember: pointers should not be used without first initializing them [both C and C++]

We will write a C program to build a linked list along with functions for insertion, deletion, search, create header etc. [experiment with it, be comfortable.]

---

**Linked Lists**

**Stacks** and **Queues** belong to a general class of data structure, called **List**. Implementing lists using arrays (contiguous memory space) is relatively easy for simple small examples – the advantage is that most key operations are O(1) due to possible indexing in arrays; the disadvantage is we cannot do insertion or deletions at arbitrary positions in O(1) time (re-compaction will take O(n) worst time) – not very effective in applications that need frequent insertions and deletions. Instead of keeping a linear list in sequential memory locations, we can make use of a much more flexible scheme (called **linked lists**) in which each node contains a link (pointer) to the next node (next element of the list).

- Linked allocation (both **singly linked** or **doubly linked**) takes up additional memory space for the links; computer memory is relatively cheap.

- It is not necessary to know the addresses of all current keys – adverse effect is searching for a specific key is of $O(n)$ complexity in the worst case – once the position is located, insertion/deletion takes $O(1)$ time.
**Linked Lists**

- The linked scheme makes it easier to join two lists together, or to break one apart into two that will grow independently [Not generally possible when using arrays].
- The linked scheme lends itself immediately to more intricate structures than simple linear lists. We can have a variable number of variable-size lists; any node of the list may be a starting point for another list; the nodes may simultaneously be linked together in several orders corresponding to different lists; and so on. The list head (a pointer, i.e., an address) points to the first element of the list.
- Read [here](#) for linked list basics; Read [here](#) for a refresher on C, and [here](#) for a refresher on C++. **Remember**: pointers should not be used without first initializing them [both C and C++]

```
head

4 ─ 3 ─ 2 ─ 1 ─ NULL
```

---

**Linked List implementation**

- **Assume**: we maintain a sorted list of integers; insertions/deletions must keep the list always sorted. We use the variable \( Lp \) (List Pointer) to be the address of the first node of the list.
- Each element in a linked list is usually called a **node** that has two components: the element and a pointer that points to the next element in the list. Use a struct.
- Pass parameters by reference if they are to be changed by a function.
- Insert will insert a new node with a key in the list in sorted order iff the key is not already present in the list. Need two pointers to scan the list – current and previous.
- Once the correct position is found with no error), create a new node and adjust the pointers accordingly (depending on if the newly inserted node is the first element of the list or not).
Linked List implementation in C

typedef struct{
    int info;
    struct node *link;
}node;

void create (node **lp){
*lp =NULL; }

void insert (node **lp, int key){
node *current, *prev, *new;
current = *lp; prev = NULL; //start at list head
while (current != NULL && current->info < key){
    prev =current;
current = current->link; }
if (current != NULL && current->info = = key)
    printf ("The key already exists; Try again\n");
else {new = (node *) malloc(sizeof(node));
new->info = key;
new->link = current;
if (prev == NULL) *lp =new
else prev->link = new; }
}

void delete (node **lp, int key){
node *current, *prev;
current = *lp; prev = NULL;
while (current != NULL && current->info < key){
    prev =current;
current = current->link; }
if (current->info == key){
    if (prev == NULL) {*lp = current->link; }
else {prev->link = current->link;
free ((void *) current);};}
else printf ("Key is not in list \n");
}

Use the functions to process a list:
node *root; insert (&root, 4); insert (&root, 5); insert (&root, 10); insert (&root, 15); insert (&root, 8); printhplease(&root); delete(&root, 10): and so on.

Exercise Problems:
- Design functions for “multi-insert”, “multi-delete”. Design a function to empty an existing list (you need to garbage collect); functions like is_empty(list).
- Design a function that inserts new elements at the end of a list (not to be sorted). Think of many others.
- Design a function to compute the length of a list.
- Next, consider a circular list, a stack, and a queue and.
- Think of doubly linked or quadruply linked lists.
- Also, consider having a “head” node to make the operations efficient.
Linked List implementation in C (contd.)

Exercise Problems:

- Design functions for “multi-insert”, “multi-delete”. Design a function to empty an existing list (you need to garbage collect); functions like is_empty(list).
- Design a function that inserts new elements at the end of a list (not to be sorted). Think of many others.
- Design a function to compute the length of a list.
- Next, consider a circular list, a stack, and a queue and.
- Think of doubly linked or quadruply linked lists (think of sparse Matrices).
- Also, consider having a “head” node to make the operations efficient
- Consider a linked list where both insertion and deletion are allowed at both ends; such structures may be called DeQueue.

Abstract Data Types (ADT)

Before we define ADTs, consider the data structures: stack (LIFO) and queue (FIFO) (we all know what they are, right?) – regular data structures like arrays or linked lists, or others, but restricted by what and how one can do with them.

Abstract Data Types (ADT): [A type is a collection of values, e.g., the Boolean type consists of the values true and false]. An ADT consists of (a) a specification of the possible values of the data type and (b) a specification of the operations that are allowed to be performed on those values in terms of the operations' inputs, outputs, and effects; ideally, one does not need to know how it is implemented. All primitive data types are ADTs; we do not need to know how integers, long integers, doubles etc. are represented as bit strings [hardware specific]. This is in conformance with SE principles of encapsulation (also known as information hiding, modularity, and separation of interface from implementation).

A distinction is made between the logical concept of a data type and its physical implementation in a computer program. For example, there are two traditional implementations for the list data type: the linked list and the array-based list.

A data structure is the implementation for an ADT. In any OO language such as C++, an ADT and its implementation together make up a class. Each operation associated with the ADT is implemented by a member function or method. The variables that define the space required by a data item are referred to as data members. An object is an instance of a class, that is, something that is created and takes up storage during the execution of a computer program.

Thus, data types have both a logical and a physical form. The definition of the data type in terms of an ADT is its logical form. The implementation of the data type as a data structure is its physical form.
**Stacks**

In a stack, the element deleted from the set is the one most recently inserted: the stack implements a last-in, first-out, or LIFO, policy. Similarly, in a queue, the element deleted is always the one that has been in the set for the longest time: the queue implements a first-in, first-out, or FIFO, policy.

The INSERT operation on a stack is often called PUSH, and the DELETE operation, which does not take an element argument, is often called POP. We can implement a stack of at most \( n \) elements with an array \( S[1 \ldots n] \). The array has an attribute \( S\.top \) that indexes the most recently inserted element; if \( S\.top = 0 \), the stack is empty. We test to see whether the stack is empty by query operation STACK-EMPTY: if we attempt to pop an empty stack, we say the stack underflows, which is normally an error; if \( S\.top \) exceeds \( n \), the stack overflows.

**Simple C++ Stack Implementation using Array**

Typically in C++, a class is defined in two files, a header (or “.h”) file and an implementation (or “.cc”) file:

```cpp
// The header file, intstack.h
class IntStack {
  public:
    IntStack(); // A constructor for this class.
    void push(int x); // The operations of the Stack ADT.
    int pop();
    void makeEmpty();
    bool isEmpty();
  private:
    int data[100]; // Contains the items on the stack.
    int top; // Number of items on the stack.
};
```

**Note:** The data for the stack is stored in the private part of the class; this ensures that if the data representation is changed, code that uses the class will not have to be modified, but has to be recompiled. (why?)

```cpp
// The file intstack.cc
#include "intstack.h"
IntStack::IntStack() {
  top = 0; // A stack is empty when it is first created.
}
void IntStack::push(int x) {
  if (top == 100) // ERROR; throw an exception.
    cerr << "Attempt to push onto a full stack";
  data[top] = x;
  top++;
}
int IntStack::pop() {
  if (top == 0) // ERROR; throw an exception
    cerr << "Attempt to pop from an empty stack";
  top--;
  return data[top];
}
void IntStack::makeEmpty() {
  top = 0;
}
bool IntStack::isEmpty() {
  return (top == 0);
}
```

**Exercise:** We cannot push more than 100 integer. Change the “push” operation so that when the stack is about to overflow, allocate a stack of twice the size, copy the original elements and deallocate the old stack (use a destructor ~IntStack); start with writing the pseudocode.
Two Simple Applications of Stacks

1. Evaluate arithmetic expressions: Say, \( x = 4 - 3 \times 2 / (7 - 4) + 8 \); Remember the precedence rules of the operators; within the same precedence class operators are executed in a left to right fashion. Idea: push each operator on the stack, but first pop and perform higher and equal precedence operations. Write the complete code.

2. Given an array \( X \), the span \( S[i] \) of \( X[i] \) is the maximum number of consecutive elements \( X[j] \) immediately preceding \( X[i] \) and such that \( X[j] \leq X[i] \). Spans have applications to financial analysis, e.g., stock at 52-week high. Say, \( X = \{6, 3, 4, 5, 2\} \), then \( S = \{1, 1, 2, 3, 1\} \); Given \( X \), compute \( S \). We will do this in details later.

An Interesting Application of Stack

1. Consider a simple arithmetic expression \( 4 - 3 \times 2 / (7 - 4) + 8 \). How are we taught to evaluate such expressions? We will assume there are only single digit operands (to avoid more compiler commands) and operators: \( +, -, *, / \) and \( ^ \) (exponentiation).

2. Operators have precedence rules: (1) \( + \) and \( - \) have equal lowest precedence, (2) \( * \) and \( / \) have equal and same precedence, (3) \( ^ \) has next higher precedence and (4) parentheses have highest precedence.

3. This kind of expressions are called infix expressions (operators are placed in between two operands).

4. One (the CPU) has to scan the expression multiple times to evaluate the expression \( \Rightarrow \) more time.

5. Consider an equivalent postfix expression: \( 4 \ 3 \ 2 \ 7 \ 4 \ 2 / \ ^ \ 8 + \)
Transform Infix to Postfix

- Create an empty stack called `opstack` for keeping operators. Create an empty list for output.
- Convert the input infix string to a list by using the string method split.
- Scan the token list (from the given infix expression) from left to right.
  - If the token is an operand, append it to the end of the output list.
  - If the token is a left parenthesis, push it on the `opstack`.
  - If the token is a right parenthesis, pop the `opstack` until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
  - If the token is an operator, ^, *, /, +, –push it on the `opstack`. However, first remove any operators already on the `opstack` that have higher or equal precedence and append them to the output list.
  - When the input expression has been completely processed, check the `opstack`. Any operators still on the stack can be removed and appended to the end of the output list.

Very Simple Amortization Example

- In an amortized analysis, we average the time required to perform a sequence of data-structure operations over all the operations performed.
- With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the average performance of each operation in the worst case.
- The amortized running time of an operation within a series of operations is the worst-case running time of the series of operations divided by the number of operations. When we perform an amortized analysis, we often gain insight into a particular data structure, and this insight can help us optimize the design.
- For now, we consider a very simple case of an array `S` with operations push `(S, x)`, pop `(S)`; we know each one operation takes O(1) time. In aggregate analysis, we show that for all `n`, a sequence of `n` operations takes worst-case time `T(n)/n` in total. In the worst case, the average cost, or amortized cost, per operation is therefore `T(n)/n` (which is obviously `O(1)` here). Note that this amortized cost applies to each operation, even when there are several types of operations in the sequence.
**Very Simple Amortization Example**

Now we add the stack operation
MULTIPOP (S, k), which removes the k top objects of stack S, popping the entire stack if the stack contains fewer than k objects. Assume that k is positive; otherwise the MULTIPOP operation leaves the stack unchanged.

Let us analyze an arbitrary sequence of n Push, Pop and Multipop operations on an initially empty stack of maximum size. In fact, although a single Multipop operation can be expensive, any sequence of n Push, Pop, and Multipop operations on an initially empty stack can cost at most O(n).

**Why?** Observe, we can pop each object from the stack at most once for each time we have pushed it onto the stack. Therefore, the number of times that Pop can be called on a nonempty stack, including calls within Multipop, is at most the number of Push operations, which is at most n. For any value of n, any sequence of n Push, Pop, and Multipop operations takes a total of O(n) time. The average cost of an operation is O(n)/n = O(1).

In such an aggregate analysis we assign the amortized cost of each operation to be the average cost. In this example, therefore, all three stack operations have an amortized cost of O(1).

MULTIPOP (S, k)
while not EMPTY_STACK(S) and k > 0
    POP (S) k--;

Note: running time is linear in the number of POP operations actually executed – if n is the maximum size of the stack, worst case for Multipop is O(n).

**Very Simple Amortization Example**

We emphasize again that although we have just shown that the average cost, and hence the running time, of a stack operation is O(1), we did not use probabilistic reasoning. We actually showed a worst-case bound of O(n) on a sequence of n operations. Dividing this total cost by n yielded the average cost per operation, or the **amortized cost**.

Another way of looking at it is: charge $2 for pushing each item; while $1 is spent to do the push, the other $ is accumulated to pay for the eventual pops (including multipops, if any). This is called the Accounting Method to compute amortized cost. Will see many more interesting examples in later courses.
Queues

Queues are ADTs that provide access at both ends, unlike the stacks. We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE. The queue has a head and a tail (alternatively, front and rear). When an element is enqueued, it takes its place at the tail of the queue, just as a newly arriving customer takes a place at the end of the line. The element dequeued is always the one at the head of the queue, like the customer at the head of the line who has waited the longest.

One way to implement a queue of at most n – 1 elements using an array Q[1… n]. The queue has an attribute Q.head that indexes, or points to, its head. The attribute Q.tail indexes the next location at which a newly arriving element will be inserted into the queue. The elements in the queue reside in locations Q.head, Q.head + 1, … Q.tail − 1, where we “wrap around” in the sense that location 1 immediately follows location n in a circular order. When Q.head = Q.tail = 1, the queue is empty. Initially, we have Q.head = Q.tail = 1. When Q.head = Q.tail+1, the queue is full. Assume n = Q.length

Enque(Q,x): {if Q.head = Q.tail+1 then exit (overflow) else {Q[Q.tail] = x; if Q.tail = Q.length then Q.tail=1 else Q.tail++}}

Dequeue(Q): {if Q.head = Q.tail = 1 then exit (underflow) else {x = Q[Q.head]; if Q.head = Q.length then Q.head = 1 else Q.head++}}

Each operation takes O(1) time.

Note that we can not use the last element of the array; why? Is there a way to eliminate that minor problem? Think!

Example and Exercise

Whereas a stack allows insertion and deletion of elements at only one end, and a queue allows insertion at one end and deletion at the other end, a deque (double-ended queue) allows insertion and deletion at both ends. Write four O(1) time procedures to insert elements into and delete elements from both ends of a deque implemented by an array (or a linked list).

How do we implement a circular queue? Can we adjust our functions (classes) to handle the overflow problem by increasing (doubling) the size of the queue?

Applications: Buffering videos in packets, waiting lists, access to shared resources, multiprogramming (e.g., round robin), auxiliary data structure for many algorithms, components of other data structures, etc.
**Doubly Linked Lists**

Use two links `prev` and `next` to facilitate traversing the list from two ends.

Instead of just a single `.next` field, each node includes both `.next` and `.previous` pointers. Insertion and deletion now require more operations, but other operations are simplified. Given a pointer to a node, insertion and deletion can be performed directly whereas in the singly linked case, the iteration typically needs to locate the point just before the point of change in the list so the `.next` pointers can be followed downstream.

---

**Some Variations**

- **Dummy Header**: Avoids the case where the head pointer is NULL. Instead, choose as a representation of the empty list a single “dummy” node whose .data field is unused or can be used to hold the length of the list or other stuff.
- **Circular**: Instead of setting the `.next` field of the last node to NULL, set it to point back around to the first node. Instead of needing a fixed head end, any pointer into the list will do if need be.
- **Tail Pointer**: The list is not represented by a single head pointer. Instead, the list is represented by a head pointer which points to the first node and a tail pointer which points to the last node. The tail pointer allows operations at the end of the list such as adding an end element or appending two lists to work efficiently.
- **Head struct**: A variant is to have a special "header" struct (a different type from the node type) which contains a head pointer, a tail pointer, and possibly a length to make many operations more efficient. Many of the reference parameter problems go away since most functions can deal with pointers to the head struct.
- **Doubly-Linked**: Instead of just a single `.next` field, each node includes both `.next` and `.previous` pointers. Insertion and deletion now require more operations, but other operations are simplified. Given a pointer to a node, insertion and deletion can be performed directly.
Circular Linked Lists

Using headers and/or making them circular

One needs to adjust the codes to accommodate the header and circularity

Things to observe:
- Lists are to be always sorted or not. Insert and delete operations may be defined in different ways.
- Headers can contain information (e.g., size of the list) about the list and/or other things.
- Implement stacks, queues, double ended queues using linked lists.

Function to reverse the linked list

Initialize three pointers prev as NULL, curr as head and next as NULL.
Iterate through the linked list. In loop, do following.
// Before changing next of current,
// store next node
next = curr->next
// Now change next of current
// This is where actual reversing happens
curr->next = prev
// Move prev and curr one step forward
prev = curr
curr = next

static void reverse(struct Node** head_ref)
{
    struct Node* prev = NULL;
    struct Node* current = *head_ref;
    struct Node* next = NULL;
    while (current != NULL)
    {
        // Store next
        next = current->next;
        // Reverse current node's pointer
        current->next = prev;
        // Move pointers one position ahead.
        prev = current;
        current = next;
    }
    *head_ref = prev;
}
Merge two sorted linked lists

- SortedMerge() should return the new list. The new list should be made by splicing together the nodes of the first two lists. For example if the first linked list a is 5->10->15 and the other linked list b is 2->3->20, then SortedMerge() should return a pointer to the head node of the merged list 2->3->5->10->15->20.
- There are few cases to deal with: either ‘a’ or ‘b’ may be empty, during processing either ‘a’ or ‘b’ may run out first, and finally, there’s the problem of starting the result list empty, and building it up while going through ‘a’ and ‘b’.
- Maintain a struct node** pointer, lastPtrRef, that always points to the last pointer of the result list. This deals with the result list when it is empty.
- Think in the following way. Assume the lists are both sorted in ascending (or descending) order. The principle is simple: the first element of the result list is the first element of ‘a’ or ‘b’; compare and we get the first element of the result and similarly for the other elements.

```c
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
    struct Node* result = NULL;
    /* point to the last result pointer */
    struct Node** lastPtrRef = &result;
    while(1)
    {
        if (a == NULL)
        {
            *lastPtrRef = b; break;
        }
        else if (b==NULL)
        {
            *lastPtrRef = a; break;
        }
        if (a->data <= b->data)
        {
            MoveNode(lastPtrRef, &a);
        }
        else
        {
            MoveNode(lastPtrRef, &b);
        }
        /* tricky: advance to point to the next "next" field */
        lastPtrRef = &((*lastPtrRef)->next);
    }
    return(result);
}
```
Exercise Problems

- Singly Linked Lists (with or without a special header node)
- Remove Duplicates if any exists
- FrontBackSplit(): Given a list, split it into two sublists — one for the front half, and one for the back half. If the number of elements is odd, the extra element should go in the front list. So FrontBackSplit() on the list {2, 3, 5, 7, 11} should yield the two lists {2, 3, 5} and {7, 11}. Easy if size is known from header (if not, compute). Or run two pointers – one slow and the other twice as fast – when fast pointer reaches the end, the slow pointer will be about half way.
- MoveNode(): This is a variant on Push(). Instead of creating a new node and pushing it onto the given list, MoveNode() takes two lists, removes the front node from the second list and pushes it onto the front of the first.
- void AlternatingSplit(node *source, node **first, node **second): takes one list and divides up its nodes to make two smaller lists. The sublists should be made from alternating elements in the original list.
- Reverse a list: Ideally, Reverse() should only need to make one pass of the list. The iterative solution is moderately complex. Recursive Reverse is quite tricky to make it scan the list only once.
- Merge two sorted lists to produce a bigger sorted list: a bit tricky if you are asked to do it in place.