MTH 4300, Lecture 22

Stacks;
Binary Trees;
Binary Search Trees

E Fink

May 5, 2025

1. Stacks

Consider the following problem:

> *You are building a Word processor. Each action performed – typing a character, deleting text, changing font style – changes the document. You want to keep track of all actions performed, in the order performed, so that you can make Ctrl-Z (undo action) work.*

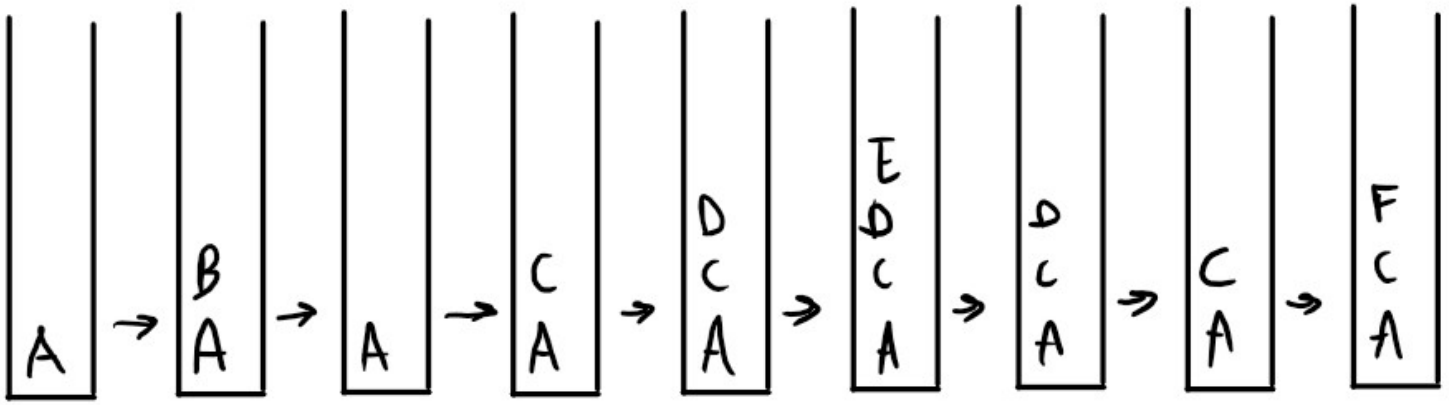
For example, if I

type A	type B	Ctrl-Z
type C	type D	type E
Ctrl-Z	Ctrl-Z	type F

what do I have on the screen at the end?

Stacks

You can visualize the "list" of actions as a pile, where the first actions are at the bottom, and further actions are added to the top – and older actions are removed from the top!



This is kind of like a list, but with a special, restricted set of operations we wish to perform on it. Note the "Last In, First Out" nature of the operations we perform.

Stacks

We can implement a class that is perfect to store actions for this type of problem: a *stack*. What a stack is can best be described presenting part of the class declaration:

```
template<typename T>
class Stack {
private: ????
public:
    Stack(): top{nullptr} {}
    void push(const T&);
    void pop();
    bool is_empty() const;
    T peek() const;
};
```

A stack is a data structure that supports the following operations:

`push()` should add a new element to the top of the stack.

`pop()` should remove the element from the top of the stack.

`peek()` should return the top element without removing it.

`is_empty()` should return `true` if there are no elements in the stack.

Stacks

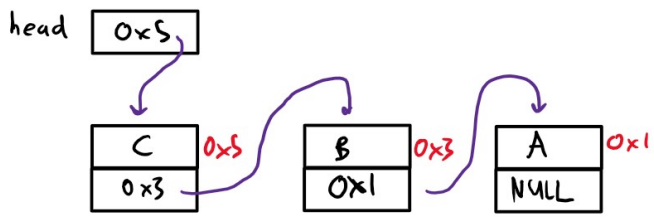
So, how can we store the elements in a stack?

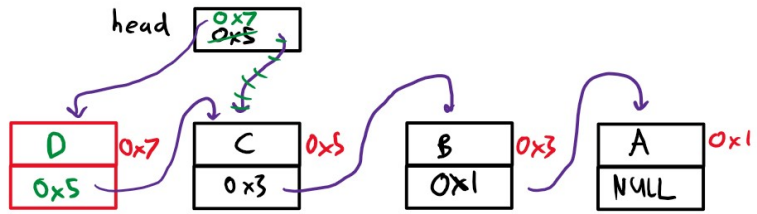
Option 1: a plain array. Big downside: absolutely fixed length.

Option 2: a vector. Small downside: periodic calls to `reserve()` are expensive.

Option 3: a linked list! (Downside: if you're not concerned about the amount of time `reserve()` takes, each typical non-resizing operation can be a little bit more expensive than with a vector)

The idea of the linked list implementation: the top of the stack is the FRONT of the linked list. After all, this element is pretty easy to add and remove. (Below is a depiction of adding the letter D to a stack already containing A, B and C.)





Stacks

So, here's a fuller class declaration:

```
template<typename T>
class Stack {
private:
    Node<T> *top;
public:
    Stack(): top{nullptr} {}
    void push(const T&);
    void pop();
    bool is_empty() const;
    T peek() const;
    // and Rule-of-3
};
```

As before, we are going to keep track of a pointer to the first `Node` of the linked list; this pointer will be called `top`.

`push()` should take the input data; create a new Node; put this new Node at the front of the linked list; and update `top` accordingly.

Let's go to **L22x1_stack.cpp** and fill out implementations.

2. Binary Trees

We've seen that linked lists allow for quick insertion and deletion, but slow look-up of values: one has to traverse the list from the beginning to get to the nth value, or to determine whether or not an element is present (even when the list is sorted).

Plain arrays are the opposite: you can access the nth element swiftly; and if the entries in the array are sorted, then you can determine whether or not an element is present in the list using binary search, which is pretty efficient. But adding and inserting elements can be costly.

Wouldn't it be great if there was a data structure that allowed fast insertion and fast look-up?

There is! A binary search tree stores information in a branching structure, that allows efficient look-up and modification.

Binary Trees

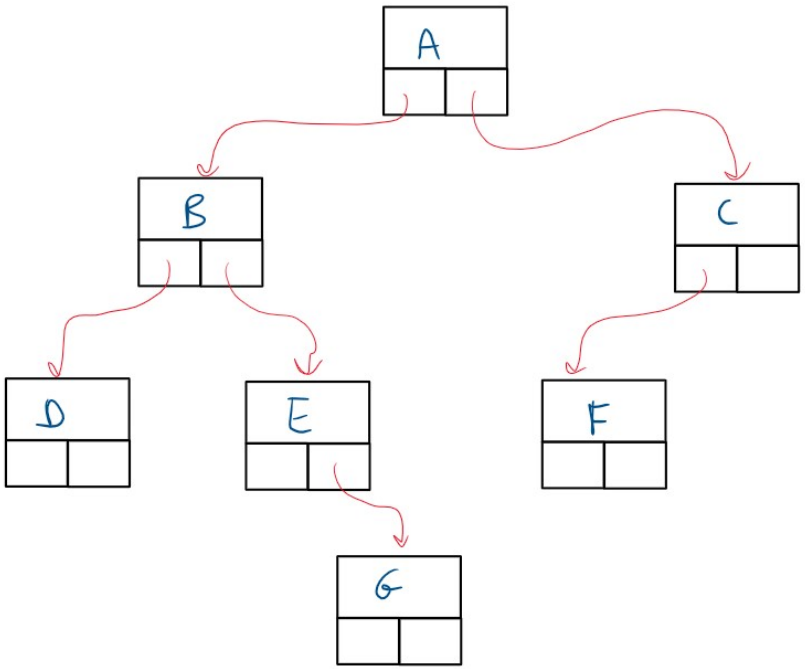
First, let's discuss binary trees in general. A binary tree, like a linked list, is composed of nodes, but these `TreeNodes` will contain TWO pointers – one to a left object and one to a right object.

```
template<typename T>
struct TreeNode {
   T data;
   TreeNode *left, *right;
   TreeNode(T d, TreeNode *l, TreeNode *r): data{d},
       left{l}, right{r} {}
};
```

Binary Trees

A binary tree will be represented by a pointer to a `TreeNode`, called the *root* of the tree. This root will contain some data, as well as pointers to two pointers to other `TreeNodes`, with one or both possibly being `nullptr`; these nodes will be referred to as *children*.

Let's encode this tree directly in `main()` in **L22x2_tree.cpp**



Binary Trees

A *leaf* of a binary tree is a node which has no children. The *height* of a binary tree would be the number of nodes in the longest direct path from the root to a leaf. For example, in the tree shown on the last page, the height would be 4, since the longest path from root to leaf is A-B-E-G, which passes through 4 nodes.

The *size* of a tree is the number of nodes it contains.

Can we code functions to compute these quantities?

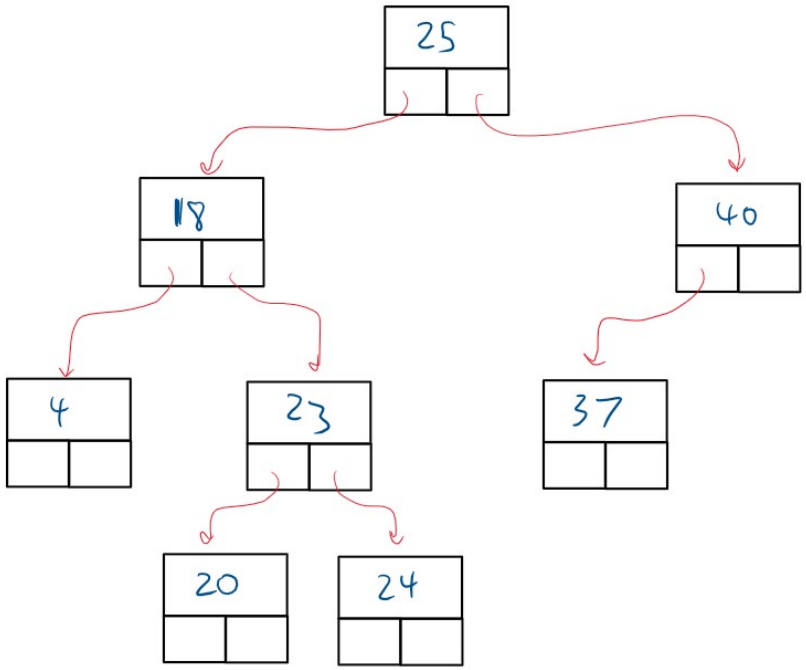
(It can be arranged that $height(T) \approx \log_2 size(T)$. This is important, since we will see that $height(T)$ is related to how long a search of a tree will take, at least if the entries in T are organized intelligently.)

3. Binary Search Trees

A binary *search* tree is a binary tree where the following property holds:

> *The data in each node is greater than or equal to the data in all the nodes of the left subtree and less than all the data in the nodes of the right subtree.*

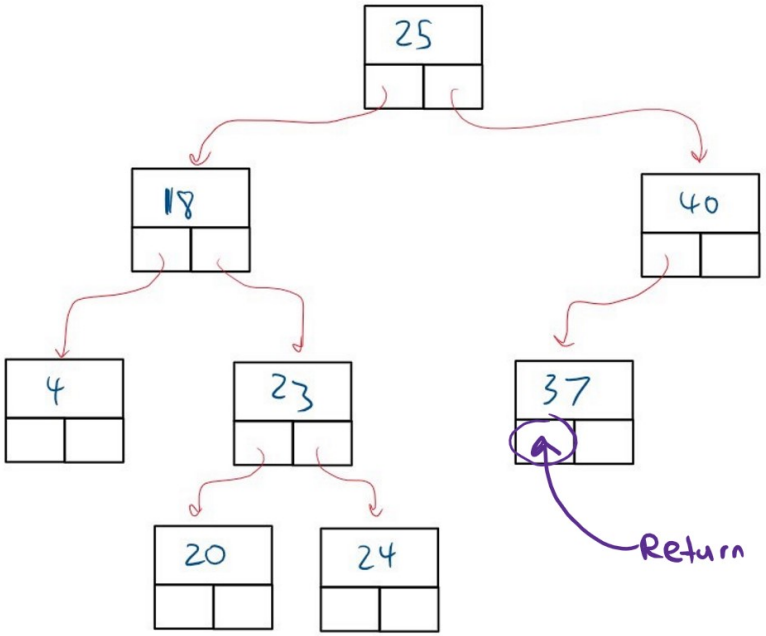
For example, the following illustrates a proper binary search tree:



Binary Search Trees

Searching for the presence of a value within a binary search tree is relatively simple:

- Compare the tree's root node data to the value sought.
- If the value is less than or equal to the root's data, search the left sub-tree, otherwise search the right sub-tree.
- If you either find the data sought or hit `nullptr`, return a pointer to the Node in question (by reference – this will be helpful for later).



For example, if we searched for 30 in this tree, we'd look at 25, then go right; then we'd look at 40, and move left; then we'd look at 37, and move left; and then we'd hit a `nullptr`, so we'd return the indicated pointer by reference. Of course, the value of that pointer will be `nullptr`.