BSTs: Deleting things

Joel Willoughby

COP 3503

Outline

Agenda

- Discuss a small number of random topics
- Memory management
- BST review

Outline

Makefiles again

- As I was grading your Makefile labs (think way back), I noticed a lot of you said that graphmenu.cpp is recompiled when graph.h was changed because it *includes* graph.h

Makefiles again

- As I was grading your Makefile labs (think way back), I noticed a lot of you said that graphmenu.cpp is recompiled when graph.h was changed because it *includes* graph.h

```
all: graph

graph: graph.o graphmenu.o
  g++ graph.o graphmenu.o -o graph

graph.o: graph.cpp graph.h
  g++ -c graph.cpp

graphmenu.o: graphmenu.cpp
  g++ -c graphmenu.cpp
```

Makefiles again

- As I was grading your Makefile labs (think way back), I noticed a lot of you said that graphmenu.cpp is recompiled when graph.h was changed because it *includes* graph.h

```
all: graph

graph: graph.o graphmenu.o
  g++ graph.o graphmenu.o -o graph

graph.o: graph.cpp graph.h
  g++ -c graph.cpp

graphmenu.o: graphmenu.cpp
  g++ -c graphmenu.cpp
```

- This is not entirely correct. It is the dependence in the makefile that causes recompilation. So, in the above, if graph.h was changed, graphmenu.cpp wouldn't be recompiled

Outline

Memory management

- The new keyword allows us to *dynamically* allocate memory for our program
- When memory is dynamically allocated, it is then up to the program to *free* it back up
- If it doesn't free up the memory it has allocated, it is called a memory leak.
- Note that all of your program's memory is reclaimed by the operating system when your program finished its execution

The delete keyword

```
1 int main(){
2     BST * bst = new BST();
3     bst->insert(1);
4     bst->insert(2);
5     // Useful stuff goes here
6     delete bst;
7 }
```

- delete takes in a pointer (just like new returns a pointer) and goes in and frees the memory being used by that pointer

The delete keyword

```
1 int main(){
2     BST * bst = new BST();
3     bst->insert(1);
4     bst->insert(2);
5     // Useful stuff goes here
6     delete bst;
7 }
```

- delete takes in a pointer (just like new returns a pointer) and goes in and frees the memory being used by that pointer
- You can only use delete on memory that has been dynamically allocated!

The delete keyword cont.

```
1 int main(){
2     BST * bst = new BST();
3     bst->insert(1);
4     bst->insert(2);
5     // Useful stuff goes here
6     delete bst;
7 }
```

- Note that the BST object has many nodes that it has dynamically allocated inside of it
- How do you suppose delete knows to delete them?

Destructors

- Whenever an object is deleted (can be static object going out of scope or dynamic object with delete), the object's *destructor* is called.
- The job of the destructor is to free up any dynamic memory that object is using.
- If you do not write a destructor, C++ uses a default destructor, which probably does not do what you want it to

```
1 BST::~BST(){
2     cout << "I am dying!!!!!" << endl;
3     // Useful destruction code here
4 }
```

A small example

```
1  class SuperArray{
2      private:
3      int * stuff;
4      public:
5      SuperArray(){
6          // Hello World! I can't wait to be useful and stuff!
7          stuff = new int[10];
8      }
9      ~SuperArray(){
10         // Goodbye cruel world!
11         // You didn't even give me any methods...
12         delete [] stuff;
13     }
14 };
```

A few more destructor notes

- You should generally avoid expicitly calling a destructor. delete will call it for you.
- It is valid to have the following:

```
1 void Something::some_function(){
2     // Do some stuff
3     delete this;
4 }
```

Be very careful doing this. If you use it, make sure you are *returning immediately* after

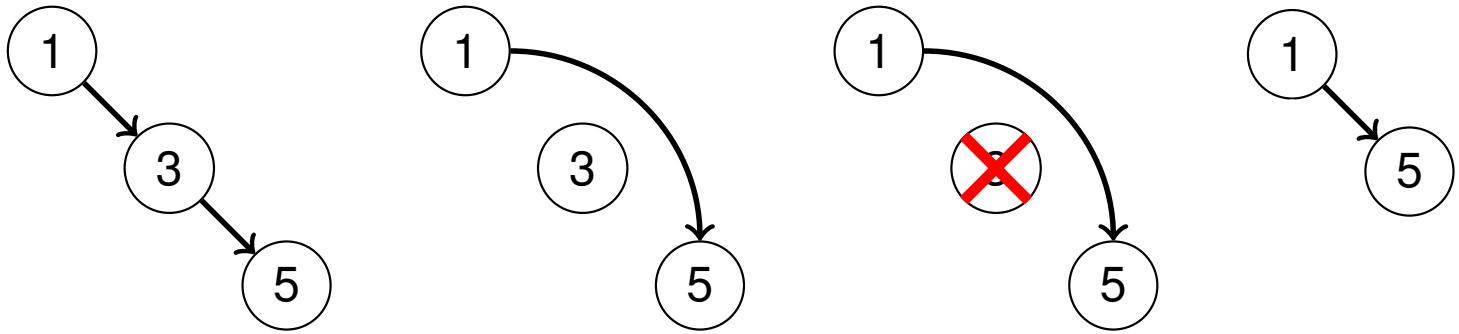
Review of BSTs

- A BST is a binary tree with the following property: Nodes to the left of the root node have values less than the root. Nodes to the right have values greater than the root.
- We have so far gone over insertion, in-order traversals, and searching.
- But what if the user wants to remove a value?

Deleting a Node

There are three cases. In each case, *We must preserve the BST property*

- Deleting a leaf node: we can just easily delete it (Don't forget to set the pointer to NULL).
- Deleting a node with one child: If this is the case, then we can do what is called splicing the node out.

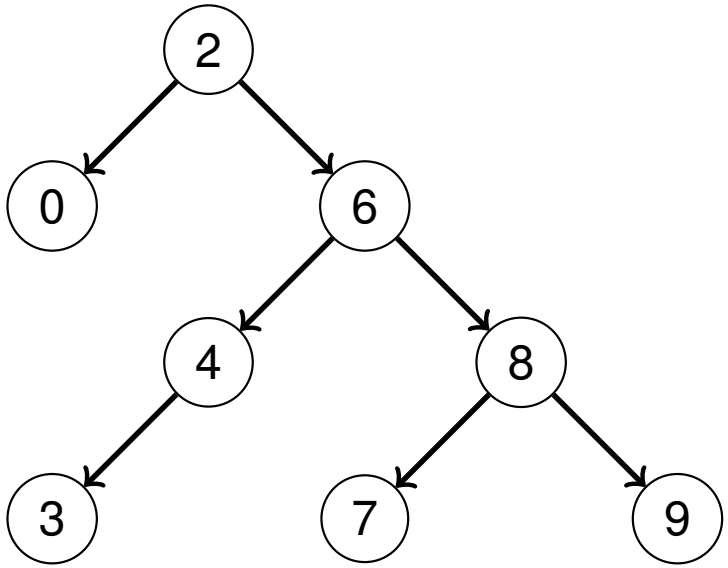


- Deleting a node with two children: This one requires a bit more thought

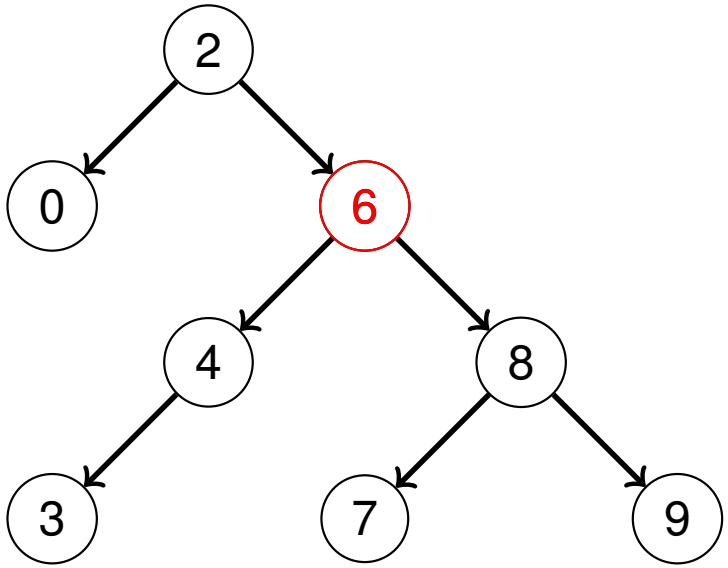
Deleting a node with 2 children

- We can't easily delete this one or splice it out.
- Instead the idea is to look in our tree and find a suitable candidate node to take its place
- We move the candidate node to the place of the one we want to delete, and then delete the old candidate node
- There are two possible nodes that are good candidates: the successor and predecessor
- These are the largest node in the left subtree and the smallest node in the right subtree.

Example

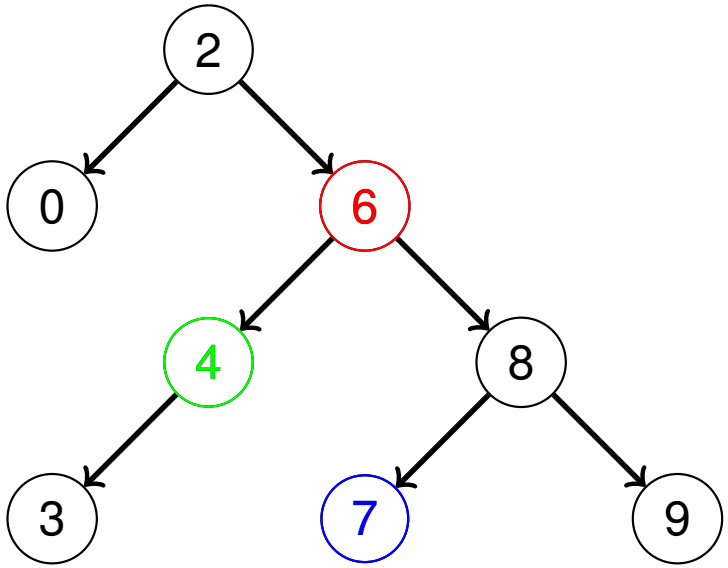


Example



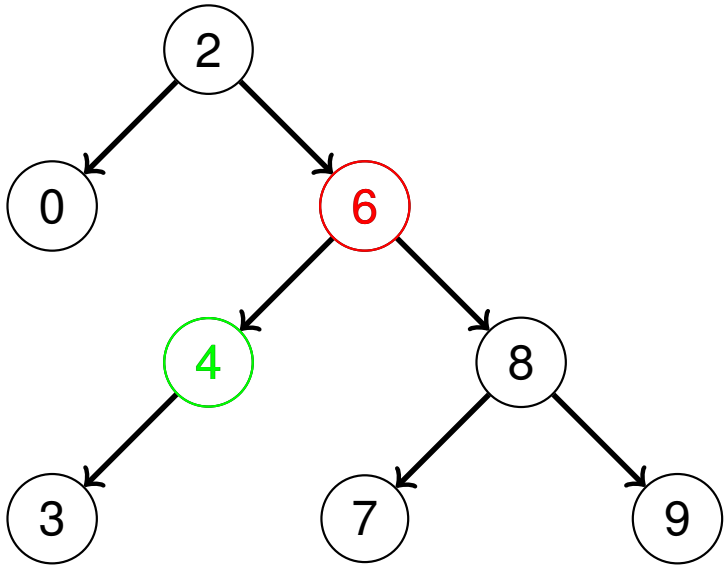
We want to remove 6

Example



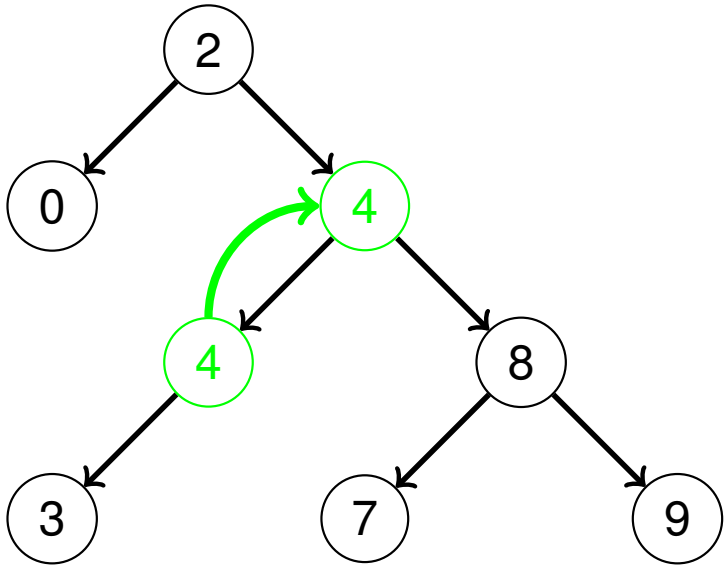
We have 2 candidates, the predecessor and the successor

Example



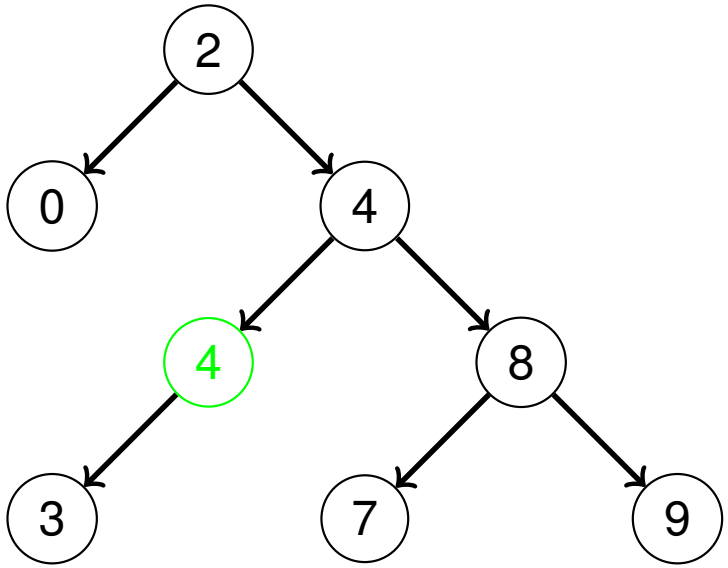
We will use the predecessor

Example



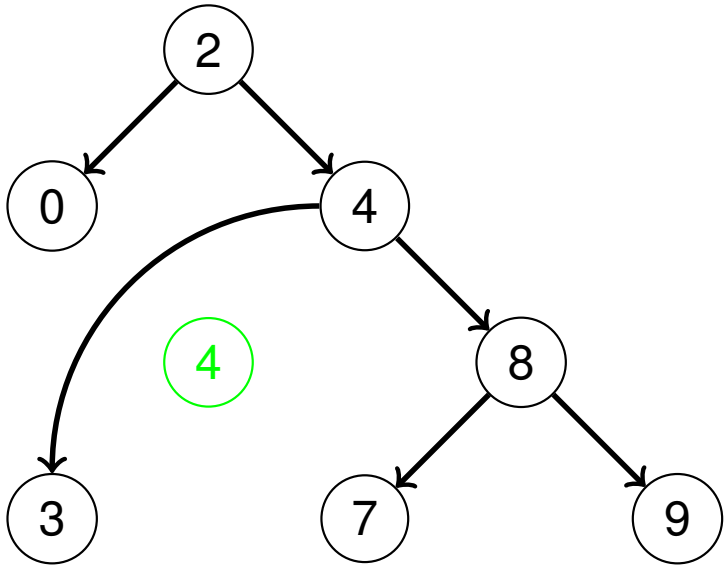
First, we copy the predecessor to take the place of the node we want to delete

Example



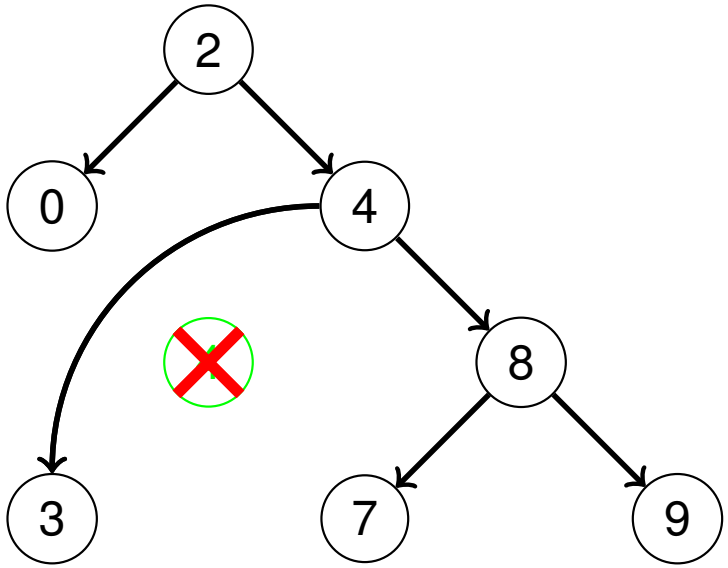
Now, we have to delete the old predecessor

Example



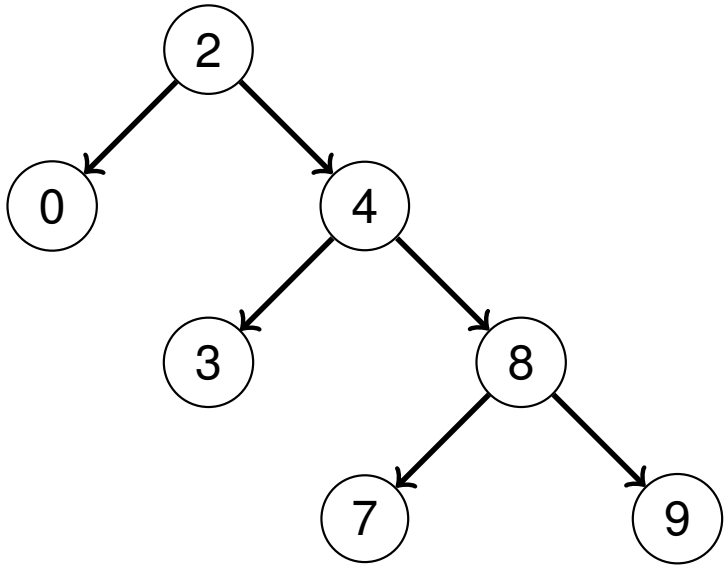
We have already done this

Example



We have already done this

Example



Yay!

Clearing a tree

- Clearing a tree can be done using a traversal
- You have already seen in-order traversals
- These won't work because you would have to delete the Node and then go to its right subtree, not a good idea...
- Instead, a post-order traversal is preferred.
- Do not forget to set your root pointer after clearing the tree!

Outline

Questions

???