Stacks

Topics:

+ Push Method
+ Pop Method
+ Top Method
+ IsEmpty Method

Resources:

- `Node.h` - `Stack.h` - `main.cpp`

Introduction

A *stack* is a data structure that follows the principle first in last out (FILO). That is, the last object added to a stack will be the first object removed from the stack. This behavior is seen in a number of virtial and real world applications especially for grammars and pattern checking. The methods of a stack are normally called `push()`, `pop()`, `top()` and `isempty()` which adds, removes, and views the top object of the stack and determines if the stack is empty respectively.

For this lecture, we will look at three implementations of a stack such that all methods of the stack that were mentioned before will have a constant big-O runtime. The implementations will be container classes of the stack implemented as an array, a singly linked list and a doubly linked list. Their fields are as follow

```
template<class T>
class Stack
{
 private:
 T* data;
 ulong top;
 ulong capacity;
};
```

```
template<class T>
class Stack
{
 private:
 Node<T>* head;
};
```

```
template<class T>
class Stack
{
 private:
 Node<T>* head;
};
```

where the leftmost is the array, the middle is the singly linked list and the rightmost is the doubly linked list. The array implementation will have a fixed size while the linked list implementations will be dynamic. Their default constructors are

```
Stack()
{
 top = 0;
 capacity = 100;
 data = new T[capacity];
}
```

```
Stack()
{
 head = NULL;
}
```

```
Stack()
{
 head = NULL;
}
```

The array implementation will have an overloaded constructor that allows the user to assign the value of the *capacity* of the stack; however, the linked link implementations will not define one. The other special member of functions (copy constructor, assignment operator and destructor) will adhere to standard implementation for their respective storage method.

Push Method

The `Push()` method adds to the stack. Since this method and the removal method should have a constant runtime, we need to determine a position of the storage method that does not require any changes or searches when performing an insertion and removal. For the array, we can add and remove from the end of the array with a constant runtime. While for the linked lists, we can add and remove from the beginning of the list with a constant time using one reference to the list. Hence, the `Push()` methods will be

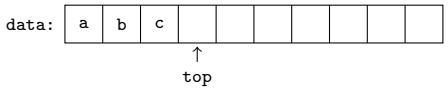
```
void Push(const T& item)
{
 if(top < capacity)
 {
  data[top] = item;
  top += 1;
 }
}
```

```
void Push(const T& item)
{
 Node<T>* t = Create(item);
 t->link = head;
 head = t;
}
```

```
void Push(const T& item)
{
 Node<T>* t = Create(item);
 t->next = head;

 if(head != NULL)
 {
  head->prev = t;
 }
 head = t;
}
```

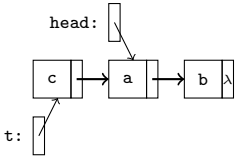
For the array implementation, *top* is used as an index of the currect end of the stack



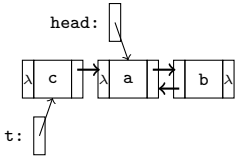
If *top* is less than *capacity*, there is space in the stack. Hence, to add an object to the end of the stack, we assign it to *data* at the index *top*; and then, we increment *top* by one to represent the new end of the stack.

For the linked list implementations, to add to the beginning of the list, we need to create a new node and make its link (or next link) equal to *head*

Singly Linked List

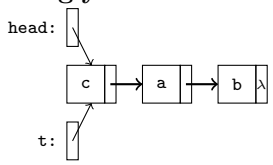


Doubly Linked List

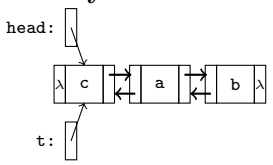


Afterwards, we need to make *head* equal to the new node. For the doubly linked list, before making *head* equal to the new node, we need the previous link of *head* to equal the new node if *head* is not referencing an empty list.

Singly Linked List



Doubly Linked List



Pop Method

The `Pop()` method removes from the stack. Its methods will be

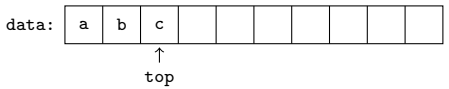
```
void Pop()
{
 if(top > 0)
 {
  top -= 1;
 }
}
```

```
void Pop()
{
 if(head != NULL)
 {
  Node<T>* t = head;
  head = head->link;
  delete t;
  t = NULL;
 }
}
```

```
void Pop()
{
 if(head != NULL)
 {
  Node<T>* t = head;
  head = head->next;
  delete t;
  t = NULL;

  if(head != NULL)
  {
   head->prev = NULL;
  }
 }
}
```

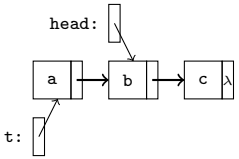
For the array implementation, if the stack is not empty, we just need to decrement *top* by 1.



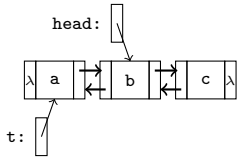
Although there is data in the previous position of *top*, it will be overridden when new data is added.

For the linked list implementations, if the stack is not empty, a new node will be assigned *head*; and then, *head* will be assigned its link (or next link).

Singly Linked List



Doubly Linked List



Afterwards, the new node will be deallocated. For the doubly linked list, the previous link of the *head* will be assigned NULL if *head* is not referencing an empty list.

Top Method

The `Top()` method views the top object of the stack, which is the object that will be removed next. For this method, you just have to check if the stack is not empty. If the stack is empty, an error is thrown; otherwise, the top object is returned. Its methods will be

```
const T& Top() const
{
 if(top == 0)
 {
  throw "Empty Stack";
 }
 return data[top-1];
}
```

```
const T& Top() const
{
 if(head == NULL)
 {
  throw "Empty Stack";
 }
 return head->data;
}
```

```
const T& Top() const
{
 if(head == NULL)
 {
  throw "Empty Stack";
 }
 return head->data;
}
```

For the array implementation, the top object is the element with the index one less than *top*. While, for the linked list implementations, the top object is the *head*.

IsEmpty Method

The `IsEmpty()` method determines if the stack is empty. This is accomplished with just a boolean expression for all implementations. Its methods will be

```
bool IsEmpty() const
{
 return (top == 0);
}
```

```
bool IsEmpty() const
{
 return (head == NULL);
}
```

```
bool IsEmpty() const
{
 return (head == NULL);
}
```

```
bool IsFull() const
{
 return (top == capacity);
}
```

The additional method `IsFull()` on the far right is for the array implementation since the stack has a fixed size. It determines if the stack is full.