CS32 Discussion
2019 Summer - Week 3
Qi Zhao
zhaoq0563@ucla.edu
Jul-12-2019
Outline

• Linked list
• Stack
• Queue
Outline

• Linked list
• Stack
• Queue
Linked Lists

- A **linked list** is a series of connected **nodes**
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- **Head**: pointer to the first node
- The last node points to **NULL**
A Simple Linked List Class

• We use two classes: **Node** and **List**
• Declare **Node** class for the nodes
  • **data**: double-type data in this example
  • **next**: a pointer to the next node in the list

```cpp
class Node {
public:
    double data; // data
    Node* next; // pointer to next
};
```
A Simple Linked List Class

• Declare **List**, which contains
  • **head**: a pointer to the first node in the list.
    Since the list is empty initially, **head** is set to **NULL**
  • Operations on **List**

```cpp
class List {
public:
    List(void)            // constructor
    ~List(void);         // destructor

    bool IsEmpty() { return head == NULL; }  
    Node* InsertNode(int index, double x);
    int FindNode(double x);
    int DeleteNode(double x);
    void DisplayList(void);

private:
    Node* head;
};
```
A Simple Linked List Class

- **Operations of List**
  - **IsEmpty**: determine whether or not the list is empty
  - **InsertNode**: insert a new node at a particular position
  - **FindNode**: find a node with a given value
  - **DeleteNode**: delete a node with a given value
  - **DisplayList**: print all the nodes in the list
Inserting a new node

- Adding a new value to the list.
Inserting a new node

1. Create a new node. Call the pointer to it \( p \).
Inserting a new node

2. Make its next pointer point to the first item.
   \[ p->next = \text{head}; \]
Inserting a new node

3. Make the head pointer point to the new node.
   head = p;
Inserting a new node

• What about insertion in the middle of the list?
• At the end of the list?
Inserting a new node

• **Possible cases of** `InsertNode`
  1. Insert into an empty list
  2. Insert in front
  3. Insert at back
  4. Insert in middle

• **But, in fact, only need to handle two cases**
  • Insert as the first node (Case 1 and Case 2)
  • Insert in the middle or at the end of the list (Case 3 and Case 4)
Inserting a new node

- Node* InsertNode(int index, double x)
  1. Locate index’th element
  2. Allocate memory for the new node
  3. Point the new node to its successor
  4. Point the new node’s predecessor to the new node
Node* List::InsertNode(int index, double x) {
    if (index < 0) return NULL;
    int currIndex = 1;
    Node* currNode = head;
    while (currNode && index > currIndex) {
        currNode = currNode->next;
        currIndex++;
    }
    if (index > 0 && currNode == NULL) return NULL;

    Node* newNode = new Node;
    newNode->data = x;

    if (index == 0) {
        newNode->next = head;
        head = newNode;
    }
    else {
        newNode->next = currNode->next;
        currNode->next = newNode;
    }
    return newNode;
}
Node* List::InsertNode(int index, double x) {
    if (index < 0) return NULL;
    int currIndex = 1;
    Node* currNode = head;
    while (currNode && index > currIndex) {
        currNode = currNode->next;
        currIndex++;
    }
    if (index > 0 && currNode == NULL) return NULL;

    Node* newNode = new Node;
    newNode->data = x;

    if (index == 0) {
        newNode->next = head;
        head = newNode;
    }
    else {
        newNode->next = currNode->next;
        currNode->next = newNode;
    }

    return newNode;
}
Node* List::InsertNode(int index, double x) {
    if (index < 0) return NULL;
    int currIndex = 1;
    Node* currNode = head;
    while (currNode && index > currIndex) {
        currNode = currNode->next;
        currIndex++;
    }
    if (index > 0 && currNode == NULL) return NULL;
    Node* newNode = new Node;
    newNode->data = x;
    if (index == 0) {
        newNode->next = head;
        head = newNode;
    } else {
        newNode->next = currNode->next;
        currNode->next = newNode;
    }
    return newNode;
}
Node* List::InsertNode(int index, double x) {
    if (index < 0) return NULL;
    int currIndex = 1;
    Node* currNode = head;
    while (currNode && index > currIndex) {
        currNode = currNode->next;
        currIndex++;
    }
    if (index > 0 && currNode == NULL) return NULL;

    Node* newNode = new Node;
    newNode->data = x;

    if (index == 0) {
        newNode->next = head;
        head = newNode;
    }
    else {
        newNode->next = currNode->next;
        currNode->next = newNode;
    }

    return newNode;
}
Finding a node

- **Node* FindNode(double x)**
  - Search for a node with the value equal to \( x \) in the list.
  - If such a node is found, return the pointer points to it. Otherwise, return NULL.

```cpp
Node* List::FindNode(double x) {
    Node* currNode = head;
    while (currNode != NULL) {
        if (currNode->data != x) {
            currNode = currNode->next;
        } else {
            return currNode;
        }
    }
    return NULL;
}
```
Deleting a node

- Suppose there is an item that you want to remove, and it is pointed by a pointer, say \texttt{p}.
- Can I just do “delete \texttt{p};”?

\begin{figure}
\centering
\begin{tikzpicture}
    \node[draw] (q) at (0,0) {q};
    \node[draw] (p) at (1,0) {p};
    \node[draw] (r) at (2,0) {r};
    \draw[->] (q) -- (p);
    \draw[->] (p) -- (r);
\end{tikzpicture}
\end{figure}

- We need to set the previous node’s (\texttt{q}) next pointer to point to the next node of \texttt{p}!
Deleting a node

- When looking up \( p \), keep the pointer to the previous node (\( q \)).

- Then ...
  
  \[ q \rightarrow \text{next} = p \rightarrow \text{next}; \]
  
  \[ \text{delete } p; \]
Deleting a node

- **Sanity Checks**
  - Does it work if \( p == \text{head} \)?
  - Does it work if \( p \) points to the last one?

\[
\text{q->next = p->next;}
\text{delete p;}
\]
Deleting a node

- If $p == \text{head}$, there is no “previous” node to $p$.
- Make an exception for this.
  - We need to reset the head pointer.

```cpp
head = p->next;
delete p;
```
```c
int List::DeleteNode(double x) {
    Node* prevNode = NULL;
    Node* currNode = head;
    int currIndex = 1;
    while (currNode && currNode->data != x) {
        prevNode = currNode;
        currNode = currNode->next;
        currIndex++;
    }
    if (currNode) {
        if (prevNode) {
            prevNode->next = currNode->next;
            delete currNode;
        } else {
            head = currNode->next;
            delete currNode;
        }
        return currIndex;
    }
    return 0;
}
```
int List::DeleteNode(double x) {
    Node* prevNode = NULL;
    Node* currNode = head;
    int currIndex = 1;
    while (currNode && currNode->data != x) {
        prevNode = currNode;
        currNode = currNode->next;
        currIndex++;
    }
    if (currNode) {
        if (prevNode) {
            prevNode->next = currNode->next;
            delete currNode;
        } else {
            head = currNode->next;
            delete currNode;
        }
        return currIndex;
    }
    return 0;
}
```cpp
int List::DeleteNode(double x) {
    Node* prevNode = NULL;
    Node* currNode = head;
    int currIndex = 1;
    while (currNode && currNode->data != x) {
        prevNode = currNode;
        currNode = currNode->next;
        currIndex++;
    }
    if (currNode) {
        if (prevNode) {
            prevNode->next = currNode->next;
            delete currNode;
        } else {
            head = currNode->next;
            delete currNode;
        }
    }
    return currIndex;
}
return 0;
```
Printing all the elements

- void DisplayList()
  - Print the data of all the elements
  - Print the number of the nodes in the list

```cpp
void List::DisplayList()
{
    int num = 0;
    Node* currNode = head;
    while (currNode != NULL){
        cout << currNode->data << endl;
        currNode = currNode->next;
        num++;
    }
    cout << "Number of nodes in the list: " << num << endl;
}
```
Destroying the list

• `~List()`
  • Use the destructor to release all the memory used by the list.
  • Step through the list and delete each node one by one.

```cpp
List::~List() {
    Node* currNode = head, *nextNode = NULL;
    while (currNode != NULL) {
        nextNode = currNode->next;
        // destroy the current node
        delete currNode;
        currNode = nextNode;
    }
}
```
What’s nice about Linked list

• Very efficient insertion
• Flexible memory allocation
  • Think about what you should do if you have to grow/shrink a dynamically allocated array
  • And yes, there is a little overhead, but that’s the price we pay
• Simple to implement
What’s **not** so nice about Linked list

- Slow search (i.e. accessing a certain element, e.g. “get the 4237th item”)
  - Usually, search is the operation that matters more than insertion or removal. (*Why?*)
Array versus Linked List

• Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  • **Dynamic**: a linked list can easily grow and shrink in size.
    • We don’t need to know how many nodes will be in the list. They are created in memory as needed.
    • In contrast, the size of a C++ array is fixed at compilation time.
  • **Easy and fast insertions and deletions**
    • To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
    • With a linked list, no need to move other nodes. Only need to reset some pointers.
Variations of Linked List

*Doubly linked lists*

- Each node points to not only successor but the predecessor.
- There are two `NULL`: at the first and last nodes in the list.

**Advantage**

- Given a node, it is easy to visit its predecessor.
- Convenient to traverse lists backwards
Insertion of DLL

Four different conditions to insert a new node P
1. Insert before head;
2. Insert after tail;
3. Insert somewhere in the middle;
4. When list is empty;
Insertion (Before head)

1) Set the *prev* of *head* to the new node *p*
   - head -> prev = p;
2) Set the *next* of *p* to *head*
   - p -> next = head;
3) *p* becomes the new *head*
   - head = p;
4) *head* -> *prev* = NULL;
Insertion (after tail)

• Quite the same as insertion before head:

```c
    tail -> next = p;
    p -> prev = tail;
    tail = p;
    p -> next = NULL;
```
Insertion in the middle (after node q)

1) Fix the next node of q first:
   - Node *r = q -> next;

2) Point both next of q and prev of r to p
   - q -> next = r -> prev = p;

3) Point both sides of p to q and r respectively:
   - p -> prev = q;
   - p -> next = r;
Insertion (to an empty list)

• How do we represent an empty list?
  • head == NULL (Or tail == NULL; Or head == tail == NULL)

• 1) Insertion, just set p as head as well as tail:
  • head = tail = p;

• 2) Don’t forget to set NULL on both sides:
  • p->next = p->prev = NULL;
Search

• Just like the singly linked list.

Node* Search(int key, Node* head) {
    Node *q = head;
    while(q != NULL) {
        if(q -> value == key) return q;
        else q = q -> next; //iterate to the next node
    }
    return NULL;
}

Node* Search(int key, Node* tail) {
    Node *q = tail;
    while(q != NULL) {
        if(q -> value == key) return q;
        else q = q -> prev; //iterate to the previous node
    }
    return NULL;
}
Removal

• More complex than singly linked list.
  • Check if the node p is the head (p == head). Let this Boolean be A.
  • Check if the node is the tail (p == tail). Let this Boolean be B.
Removal

Four cases:

• Case 1 (A, but not B): P is the head of the list, and there is more than one node.
• Case 2 (B, but not A): P is the tail of the list, and there is more than one node.
• Case 3 (A and B): P is the only node.
• Case 4 (not A and not B): P is in the middle of the list.
Removal Case 1 (P is head)

• 1) Update head
  • head = head -> next;

• 2) delete p
  • delete p;

• 3) Set the prev of head to NULL
  • head -> prev = NULL;
Removal Case 2 (P is tail)

• 1) Update tail
  • tail = tail -> prev;

• 2) delete p;

• 3) Set the next of tail to NULL
  • tail -> next = NULL;
Removal Case 3 (P is the only node)

• 1) Empty the linked list:
  • head = tail = NULL;
• 2) delete p;
Removal Case 4 (P is in the middle)

• 1) Fix the prev and next of p:
  • Node *q = p -> prev;
  • Node *r = p -> next;

• 2) Concatenate q and r:
  • q -> next = r;
  • r -> prev = q;

• 3) Delete p;
Removal Case 4 (Equivalent implementation)

• If we do not fix with q and r:
  • p -> prev -> next = p -> next;
  • p -> next -> prev = p -> prev;
  • delete p;
void removeNodeInDLL(Node *p, Node& *head, Node& *tail) {
    if (p == head && p == tail) //case 3
        head = tail = NULL;
    else if (p == head) { //case 1
        head = head -> next;
        head -> prev = NULL;
    }
    else if (p == tail) { //case 2
        tail = tail -> prev;
        tail -> next = NULL;
    }
    else { //case 4
        p -> prev -> next = p -> next;
        p -> next -> prev = p -> prev;
    }
    delete p;
}
Copying a doubly linked list

• 1) Create head and tail for the new list
• 2) Iterate through the old list. For each node, copy its value to a new node.
• 3) Insert the new node to the tail of the new list.
• 4) Repeat 3~4 until we have iterated the entire old list. Set NULL before head and next to tail.
void copyDDL(Node *head_o, Node *tail_o, Node& *head_n, Node& *tail_n) {
    if (tail_o == NULL) { // the original list is empty
        head_n = tail_n = NULL; return;
    }
    Node *q = head_o; // iterator
    Node *p = new Node();
    p -> value = q -> value;
    head_n = tail_n = p;
    q = q -> next;
    while (q) {
        p = new Node();
        p -> value = q -> value;
        tail_n -> next = p;
        p -> prev = tail_n;
        tail_n = tail_n -> next;
        q = q -> next;
    }
    head_n -> prev = tail_n -> next = NULL
}
Cautions about coding with a linked list

• To draw diagrams of nodes will be extremely helpful.
• When copying a linked list, only copy values to new nodes. Do not copy pointers.
Example Problem

Suppose you have a struct `Node` and a class `LinkedList` defined as follows:

```cpp
class LinkedList {
public:
    void rotateLeft(int n); // rotates head left by n
    // Other working functions such as insert and printItems
private:
    Node* head;
}
```

Give a definition for the `rotateLeft` function such that it rotates the linked list represented by `head` left by `n`. Rotating a list left consists of shifting elements left, such that elements at the front of the list loop around to the back of the list. The new start of the list should be stored within `head`.

Ex: Suppose you have a `LinkedList` object `numList`, and printing out the values of `numList` gives the following output, with the head pointing to the node with 10 as its value:

```
10 -> 1 -> 5 -> 2 -> 1 -> 73
```

Calling `numList.rotateLeft(3)` would alter `numList`, so that printing out its values gives the following, new output, with the head storing 2 as its value:

```
2 -> 1 -> 73 -> 10 -> 1 -> 5
```

The `rotateLeft` function should accept only integers greater than or equal to 0. If the input does not fit this requirement, it may handle the case in whatever reasonable way you desire.
Example Solution

```cpp
void LinkedList::rotateLeft(int n) {
    if (head == nullptr) //empty linked list
        return;
    int size = 1;
    Node* oldTail = head;
    while (oldTail->next != nullptr) {
        size++; //calculate size of the linked list
        oldTail = oldTail->next; //oldTail points to last node of linked list
    }
    if (n % size > 0) {// check if valid n was given
        int headPos = n % size;
        Node* newTail = head;
        for (int x = 0; x < headPos - 1; x++) {
            newTail = newTail->next;
        }
        Node* newHead = newTail->next;
        newTail->next = nullptr;
        oldTail->next = head;
        head = newHead; //set the head to newHead appropriately
    }
}
```
Outline

• Linked list
• Stack
• Queue
Stack (FILO)

class Stack
{
    public:
        bool push(const ItemType& item); // true if successful
        ItemType pop(); // pop
        bool empty() const; // true if empty
        int count() const; // number of items
    private:
        // Some data structure that keeps the items.
};
Applications of Stack

• Stack memory
  • It's a special region of your computer's memory that stores temporary variables created by each function (including the main() function).
  • That’s how functions work.

• Some calculators use stacks for performing mathematical operations.

• Depth-first-search
Depth-first-search

Task: Conduct a depth-first search of the graph starting with node D
Walk-Through

The order nodes are visited:

D
Walk-Through

The order nodes are visited:

D

Consider nodes adjacent to D, decide to visit C first (Rule: visit adjacent nodes in alphabetical order)
Walk-Through

The order nodes are visited:

D, C
Walk-Through

The order nodes are visited:
D, C

No nodes adjacent to C; cannot continue $\Rightarrow$ backtrack, i.e., pop stack and restore previous state
Walk-Through

The order nodes are visited:
D, C

Back to D – C has been visited, decide to visit E next
Walk-Through

The order nodes are visited:

D, C, E

Back to D – C has been visited, decide to visit E next
Walk-Through

The order nodes are visited:
D, C, E

Only G is adjacent to E
Walk-Through

The order nodes are visited:
D, C, E, G
The order nodes are visited: D, C, E, G

Nodes D and H are adjacent to G. D has already been visited. Decide to visit H.
Walk-Through

The order nodes are visited:
D, C, E, G, H

Visit H
Walk-Through

The order nodes are visited:
D, C, E, G, H

Nodes A and B are adjacent to F.
Decide to visit A next.
Walk-Through

The order nodes are visited:
D, C, E, G, H, A
Walk-Through

The order nodes are visited:
D, C, E, G, H, A

Only Node B is adjacent to A.
Decide to visit B next.

Visited Array

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>√</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Walk-Through

The order nodes are visited:
D, C, E, G, H, A, B

Visited Array

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>✓</td>
</tr>
<tr>
<td>B</td>
<td>✓</td>
</tr>
<tr>
<td>C</td>
<td>✓</td>
</tr>
<tr>
<td>D</td>
<td>✓</td>
</tr>
<tr>
<td>E</td>
<td>✓</td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>✓</td>
</tr>
<tr>
<td>H</td>
<td>✓</td>
</tr>
</tbody>
</table>

Visit B
Walk-Through

No unvisited nodes adjacent to B. Backtrack (pop the stack).

The order nodes are visited:

D, C, E, G, H, A, B
Walk-Through

The order nodes are visited:
D, C, E, G, H, A, B

No unvisited nodes adjacent to
A. Backtrack (pop the stack).
Walk-Through

The order nodes are visited:
D, C, E, G, H, A, B

No unvisited nodes adjacent to H. Backtrack (pop the stack).

Visited Array

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>√</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Walk-Through

The order nodes are visited:
D, C, E, G, H, A, B

No unvisited nodes adjacent to G. Backtrack (pop the stack).
The order nodes are visited:
D, C, E, G, H, A, B

No unvisited nodes adjacent to E. Backtrack (pop the stack).
The order nodes are visited:
D, C, E, G, H, A, B

F is unvisited and is adjacent to D. Decide to visit F next.
Walk-Through

The order nodes are visited:
D, C, E, G, H, A, B, F

Visited Array

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>√</td>
</tr>
<tr>
<td>B</td>
<td>√</td>
</tr>
<tr>
<td>C</td>
<td>√</td>
</tr>
<tr>
<td>D</td>
<td>√</td>
</tr>
<tr>
<td>E</td>
<td>√</td>
</tr>
<tr>
<td>F</td>
<td>√</td>
</tr>
<tr>
<td>G</td>
<td>√</td>
</tr>
<tr>
<td>H</td>
<td>√</td>
</tr>
</tbody>
</table>

Visit F
Walk-Through

The order nodes are visited:
D, C, E, G, H, A, B, F

No unvisited nodes adjacent to F. Backtrack.
Walk-Through

The order nodes are visited:
D, C, E, G, H, A, B, F

Visited Array

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>✓</td>
</tr>
<tr>
<td>B</td>
<td>✓</td>
</tr>
<tr>
<td>C</td>
<td>✓</td>
</tr>
<tr>
<td>D</td>
<td>✓</td>
</tr>
<tr>
<td>E</td>
<td>✓</td>
</tr>
<tr>
<td>F</td>
<td>✓</td>
</tr>
<tr>
<td>G</td>
<td>✓</td>
</tr>
<tr>
<td>H</td>
<td>✓</td>
</tr>
</tbody>
</table>

No unvisited nodes adjacent to D. Backtrack.
Walk-Through

The order nodes are visited:
D, C, E, G, H, A, B, F

Stack is empty. Depth-first traversal is done.

<table>
<thead>
<tr>
<th>Visited Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>G</td>
</tr>
<tr>
<td>H</td>
</tr>
</tbody>
</table>
Depth-first-search

DFS(G,v)  (v is the vertex where the search starts)
Stack S := {};  (start with an empty stack)
for each vertex u, set visited[u] := false;
push S, v;
while (S is not empty) do
    u := pop S;
    if (not visited[u]) then
        visited[u] := true;
        for each unvisited neighbour w of u
            push S, w;
    end if
end while
END DFS()
Implementation of Stacks

- Container: *linked list*, (or *dynamic array*).
- If linked list:
  - Push: Insert node before head.
  - Pop: remove head.
  - Top: read head.
- Count() \ size(): online maintain (with a member variable).
**Infix to Postfix**

- **Algorithm**
  1. Scan the infix expression from left to right.
  2. If the scanned character is an operand, output it.
  3. Else,
     .....3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty), push it.
     .....3.2 Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.
  4. If the scanned character is an ‘(‘, push it to the stack.
  5. If the scanned character is an ‘)’, pop and output from the stack until an ‘(‘ is encountered.
  6. Repeat steps 2-6 until infix expression is scanned.
  7. Pop and output from the stack until it is not empty.
Postfix Evaluation

1) Create a stack to store operands (or values).
2) Scan the given expression and do following for every scanned element.
   .....a) If the element is a number, push it into the stack
   .....b) If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
3) When the expression is ended, the number in the stack is the final answer
Outline

• Linked list
• Stack
• Queue
Queue (FIFO)

class Queue
{
    public:
        bool enqueue(const ItemType& item); // push
        ItemType dequeue(); // pop
        bool empty() const; // true if empty
        int count() const; // number of items
    private:
        // some data structure that keeps the items
};
The Queue Operations

- A queue is like a line of people waiting for a bank teller. The queue has a **front** and a **rear**.
The Queue Operations

- New people must enter the queue at the rear. The C++ queue class calls this a push, although it is usually called an enqueue operation.
The Queue Operations

• When an item is taken from the queue, it always comes from the front. The C++ queue calls this a pop, although it is usually called a dequeue operation.
A queue can be implemented with an array, as shown here. For example, this queue contains the integers 4 (at the front), 8 and 6 (at the rear).

An array of integers to implement a queue of integers

We don't care what's in this part of the array.
Array Implementation

- The easiest implementation also keeps track of the number of items in the queue and the index of the first element (at the front of the queue), the last element (at the rear).

[0] [1] [2] [3] [4] [5] ...

4 8 6
Dequeue Operation

- When an element leaves the queue, size is decremented, and first changes, too.
Enqueue Operation

- When an element enters the queue, size is incremented, and last changes, too.
At the End of the Array

- There is special behavior at the end of the array. For example, suppose we want to add a new element to this queue, where the last index is [5]:

```plaintext
[ 0 ] [1] [ 2 ] [ 3 ] [ 4 ] [ 5 ]
```

- Size: 3
- First: 3
- Last: 5
At the End of the Array

- The new element goes at the front of the array (if that spot isn’t already used):

```
[ 0 ] [1] [ 2 ] [ 3 ] [ 4 ] [ 5 ]
4   [ ] 2   6   1
```

- `size`: 4
- `first`: 3
- `last`: 0
Array Implementation

- Easy to implement
- But it has a limited capacity with a fixed array
- Or you must use a dynamic array for an unbounded capacity
- Special behavior is needed when the rear reaches the end of the array.
Linked List Implementation

• A queue can also be implemented with a linked list with both a head and a tail pointer.
Linked List Implementation

• Which end do you think is the front of the queue? Why?
Linked List Implementation

• The head_ptr points to the front of the list.
• Because it is harder to remove items from the tail of the list.
Applications of Queues

• Windowed data streams.
• Process scheduling (Round Robin)
• Breadth-first-search
Breadth-first-search
Breadth First Search
Breadth First Search

enqueue source node

front A

FIFO Queue

graph:
A --- B --- C
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
E --- F --- G --- H
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
I --- A


Breadth First Search

Breadth First Search

Forward

FIFO Queue

front

A

dequeue next vertex

A

FIFO Queue
Breadth-First Search

visit neighbors of A

front

FIFO Queue
Breadth First Search

visit neighbors of A

front

FIFO Queue
Breadth First Search

B discovered

front FIFO Queue
Breadth First Search

visit neighbors of A

front B

FIFO Queue
Breadth First Search

I discovered

front: B, I

FIFO Queue
Breadth First Search

finished with A

front B I

FIFO Queue
Breadth First Search

dequeue next vertex

front: B I

FIFO Queue
Breadth-First Search

visit neighbors of B

front I

FIFO Queue
Breadth First Search

visit neighbors of B

front I

FIFO Queue
Breadth First Search

F discovered

front: I F

FIFO Queue
Breadth-First Search

A

B

visit neighbors of B

front: I F

FIFO Queue
Breadth First Search

A already discovered

front: I F

FIFO Queue
Breadth First Search

finished with B

front

I F

FIFO Queue
Breadth First Search
Breadth First Search

visit neighbors of I

front F

FIFO Queue
Breadth First Search

visit neighbors of I

front FIFO Queue
Breadth First Search

A already discovered

front F

FIFO Queue
Breadth First Search

visit neighbors of I

front F

FIFO Queue
Breadth First Search

E discovered

front

FIFO Queue
Breadth First Search

- Visit neighbors of I
- Front: F E
- FIFO Queue

Diagram:
- Nodes: A, B, C, D, E, F, G, H, I
- Edges:
  - A to I
  - I to B
  - B to F
  - F to I
  - I to E
  - E to A
  - C to D
  - C to G
  - G to H
Breadth First Search

A - E - F - I
F already discovered

front: F E

FIFO Queue
Breadth First Search

I finished

front FIFO Queue
Breadth First Search

dequeue next vertex

front  F  E

FIFO Queue
Breadth First Search

visit neighbors of F

front E

FIFO Queue
Breadth First Search

G discovered

front

E G

FIFO Queue
Breadth First Search

A

B

C

D

E

F

G

H

I

A

E

F

G

I

front

FIFO Queue

F finished
Breadth First Search

dequeue next vertex

front: E G
FIFO Queue
Breadth First Search

visit neighbors of E

front G FIFO Queue
Breadth First Search

A

B

C

D

E

F

G

H

I

A

E finished

front

G

FIFO Queue
Breadth First Search

dequeue next vertex
front
FIFO Queue
Breadth First Search

visit neighbors of G

front

FIFO Queue
**Breadth First Search**

A Breadth First Search algorithm processes nodes in a breadth-first manner. It starts at the root node (in this case, A) and explores all the neighboring nodes at the present depth before moving on to nodes at the next depth level. The search continues until all nodes have been visited.

- **C discovered**: This indicates that node C has been discovered during the search process.
- **front**: The front of the FIFO queue shows the next node to be explored.
- **FIFO Queue**: This represents the queue used to store the nodes that have been visited but not yet explored in a breadth-first manner.
Breadth First Search

visit neighbors of G

front C

FIFO Queue
Breadth First Search

A

B

C

D

E

F

G

H

I

A

B

C

D

E

F

G

H

I

H discovered

front

FIFO Queue

C

H
Breadth First Search

G finished

front C H

FIFO Queue
Breadth First Search

dequeue next vertex

front FIFO Queue
Breadth First Search

- Visit neighbors of C

- FIFO Queue
Breadth First Search

D discovered
front: H D
FIFO Queue
Breadth First Search

- Breadth First Search
- FIFO Queue
- A finished
Breadth First Search

get next vertex

front: H D

FIFO Queue
Breadth First Search

visit neighbors of H

front D

FIFO Queue
Breadth First Search

Breadth First Search

A

B

C

D

E

F

G

H

I

finished H

front

D

FIFO Queue
Breadth First Search

dequeue next vertex

front D

FIFO Queue
Breadth First Search

visit neighbors of D

front

FIFO Queue
Breadth-First Search

D finished

FIFO Queue
Breadth First Search

A

B

C

D

E

F

G

H

I

enqueue next vertex

front

FIFO Queue
Breadth First Search

A - B - F - G - C - D
I - E - F - G - H

STOP

FIFO Queue

front
procedure BFS(G, root):
    Q := queue initialized with {root}
    while Q is not empty:
        current = Q.dequeue()
        if current is the goal:
            return current
        for each node n that is adjacent to current:
            if n is not labeled as discovered:
                label n as discovered
                n.parent = current
                Q.enqueue(n)
Question

• How to implement a queue with two stacks?
class Queue<E>
{
    public:
    void enqueue(E item) {
        inbox.push(item);
    }

    E dequeue() {
        if (outbox.isEmpty()) {
            while (!inbox.isEmpty()) {
                outbox.push(inbox.top());
                inbox.pop();
            }
        }
        E rst = outbox.top()
        outbox.pop();
        return rst;
    }

    private:
    Stack<E> inbox, outbox;
};
Stack, Queue in C++ STL

• http://www.cplusplus.com/reference/stack/stack/
• http://www.cplusplus.com/reference/queue/queue/
See you next week!