CSCI 104
Queues and Stacks

Mark Redekopp
David Kempe

ARRAY-BASED LIST IMPLEMENTATIONS

BOUNDED DYNAMIC ARRAY STRATEGY

A Bounded Dynamic Array Strategy

- Allocate an array of some user-provided size
  - Capacity is then fixed
- What data members do I need?

- Together, think through the implications of each operation when using a bounded array (what issues could be caused due to it being bounded)?

```
#ifndef BALISTINT_H
#define BALISTINT_H

class BAListInt {
 public:
  BAListInt(unsigned int cap);

  bool empty() const;
  unsigned int  size() const;
  void insert(int pos,
              const int& val);
  void remove(int pos);
  int const & get(int loc) const;
  int& get(int loc);
  void set(int loc, const int& val);
  void push_back(const int& val);
 private:



};
#endif
```

balistint.h

A Bounded Dynamic Array Strategy

- What data members do I need?
  - Pointer to Array
  - Current size
  - Capacity
- Together, think through the implications of each operation when using a static (bounded) array
  - Push_back: Run out of room?
  - Insert: Run out of room, invalid location

```
#ifndef BALISTINT_H
#define BALISTINT_H

class BAListInt {
 public:
  BAListInt(unsigned int cap);

  bool empty() const;
  unsigned int  size() const;
  void insert(int pos,
              const int& val);
  void remove(int pos);
  int const & get(int loc) const;
  int& get(int loc);
  void set(int loc, const int& val);
  void push_back(const int& val);
 private:
  int* data_;
  unsigned int size_;
  unsigned int cap_;

};
#endif
```

balistint.h

Implementation

- Implement the following member functions
  - A picture to help write the code

0	1	2	3	4	5	6	7
30	51	52	53	54	10		

```
BAListInt::BAListInt (unsigned int cap)
{


}

void BAListInt::push_back(const int& val)
{






}

void BAListInt::insert(int loc, const int& val)
{






}
```

balistint.cpp

Implementation (cont.)

- Implement the following member functions
  - A picture to help write the code

0	1	2	3	4	5	6	7
30	51	52	53	54	10		

```
void BAListInt::remove(int loc)
{



}
```

balistint.cpp

Array List Runtime Analysis

- What is worst-case runtime of set(i, value)?
- What is worst-case runtime of get(i)?
- What is worst-case runtime of pushback(value)?
- What is worst-case runtime of insert(i, value)?
- What is worst-case runtime of remove(i)?

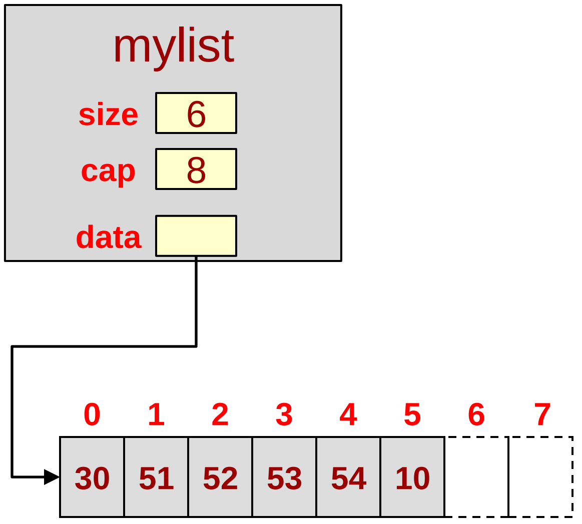
Const-ness

- Notice the get() functions?
- Why do we need two versions of get?
- Because we have two use cases…
  - 1. Just read a value in the array w/o changes
  - 2. Get a value w/ intention of changing it

```
#ifndef BALISTINT_H
#define BALISTINT_H

class BAListInt {
 public:
  BAListInt(unsigned int cap);

  bool empty() const;
  unsigned int  size() const;
  void insert(int pos, const int& val);
  bool remove(int pos);

  int& const  get(int loc) const;
  int& get(int loc);

  void set(int loc, const int& val);
  void push_back(const int& val);
 private:



};
#endif
```

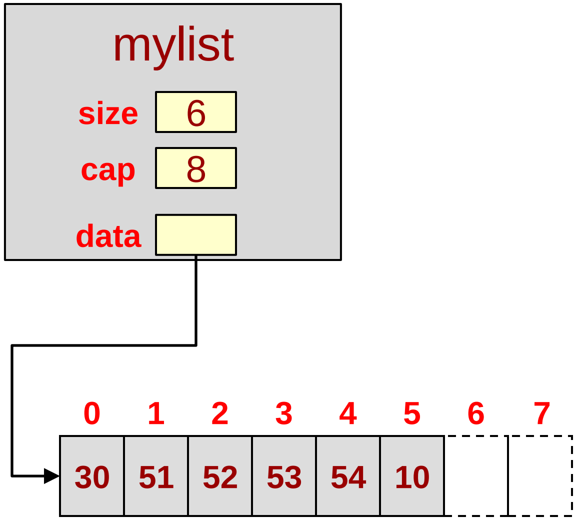
Constness

```
// ---- Recall List Member functions ------
// const version
int& const  BAListInt::get(int loc) const
{ return data_[i]; }

// non-const version
int& BAListInt::get(int loc)
{ return data_[i]; }

void BAListInt::insert(int pos, const int& val);


// ---- Now consider this code ------
void f1(const BAListInt& mylist)
{
  // This calls the const version of get.
  // W/o the const-version this would not compile
  //  since mylist was passed as a const parameter
  cout << mylist.get(0) << endl;
  mylist.insert(0, 57); // won't compile..insert is non-const

}

int main()
{
  BAListInt mylist;
  f1(mylist);

}
```



Returning References

```
// ---- Recall List Member functions ------
// const version
int& const  BAListInt::get(int loc) const
{ return data_[i]; }

// non-const version
int& BAListInt::get(int loc)
{ return data_[i]; }

void BAListInt::insert(int pos, const int& val);

// ---- Now consider this code ------
void f1(BAListInt& mylist)
{
  // This calls the non-const version of get
  // if you only had the const-version this would not compile
  //  since we are trying to modify what the
  //  return value is referencing
  mylist.get(0) += 1;  // mylist.get(0) = mylist.get(0) + 1;
  mylist.insert(0, 57);
  // will compile since mylist is non-const
}
int main()
{ BAListInt mylist;
  f1(mylist);
}
```

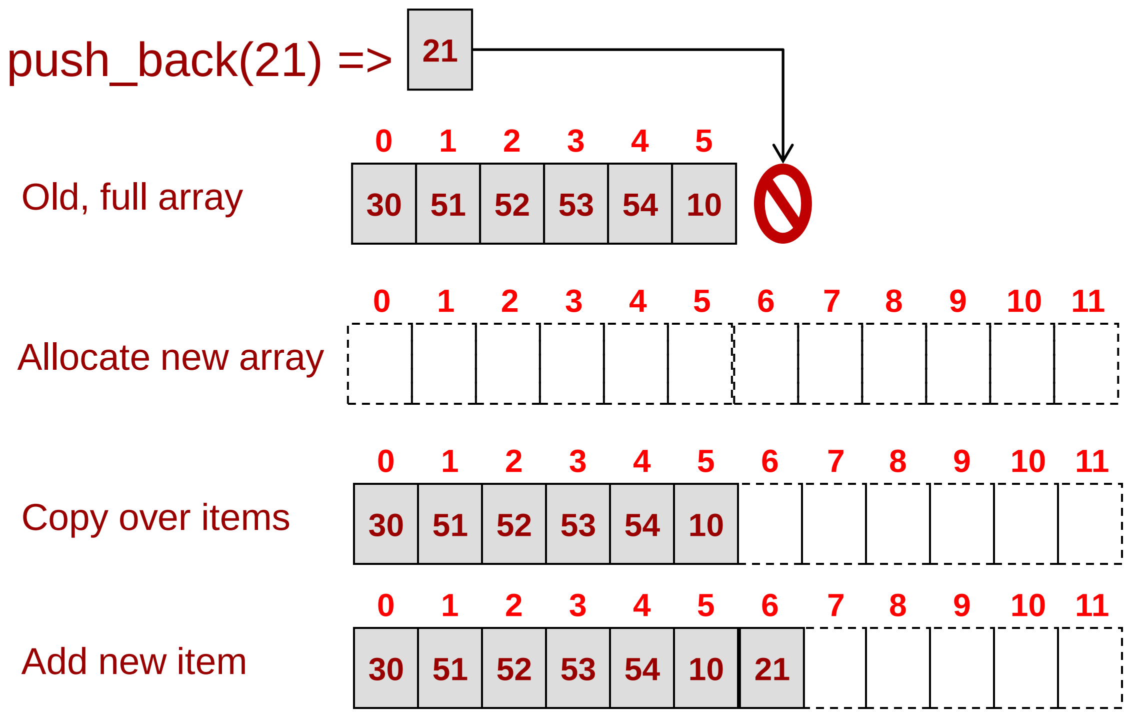


Moral of the Story: We need both versions of get()

UNBOUNDED DYNAMIC ARRAY STRATEGY

Unbounded Array

- Any bounded array solution runs the risk of running out of room when we insert() or push_back()
- We can create an unbounded array solution where we allocate a whole new, larger array when we try to add a new item to a full array



We can use the strategy of allocating a new array ***twice*** the size of the old array

Activity

- What function implementations need to change if any?

```
#ifndef ALISTINT_H
#define ALISTINT_H

class AListInt  {
 public:
  bool empty() const;
  unsigned int  size() const;
  void insert(int loc, const int& val);
  void remove(int loc);
  int& const   get(int loc) const;
  int& get(int loc);
  void set(int loc, const int& val);
  void push_back(const T& new_val);
 private:


  int* _data;
  unsigned int _size;
  unsigned int _capacity;
};

// implementations here
#endif
```

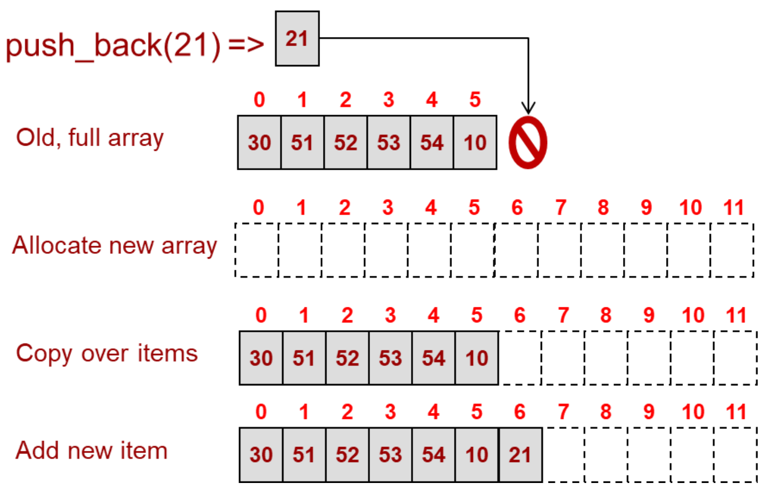
Activity

- What function implementations need to change if any?

```
#ifndef ALISTINT_H
#define ALISTINT_H

class AListInt  {
 public:
  bool empty() const;
  unsigned int  size() const;
  void insert(int loc, const int& val);
  void remove(int loc);
  int& const   get(int loc) const;
  int& get(int loc);
  void set(int loc, const int& val);
  void push_back(const T& new_val);
 private:
  void resize(); // increases array size
  int* _data;
  unsigned int _size;
  unsigned int _capacity;
};

// implementations here
#endif
```

Resizing

- Implement the resize method for an unbounded dynamic array




```
#include "alistint.h"

void AListInt::resize()
{

}
```

alistint.cpp

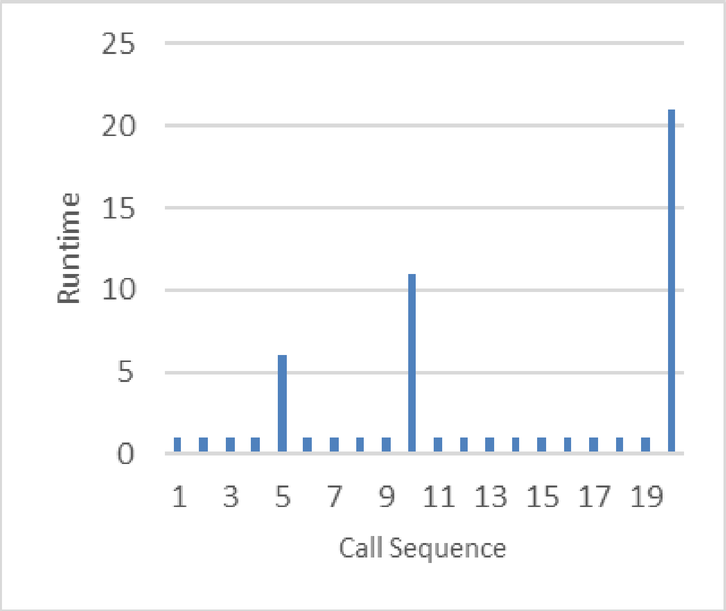
A LOOK AHEAD: AMORTIZED RUNTIME

Example

- You love going to Disneyland. You purchase an annual pass for $240. You visit Disneyland once a month for a year. Each time you go you spend $20 on food, etc.
  - What is the cost of a visit?
- Your annual pass cost is spread or "**amortized**" (or averaged) over the duration of its usefulness
- Often times an operation on a data structure will have similar "irregular" (i.e. if we can prove the worst case can't happen each call) costs that we can then amortize over future calls

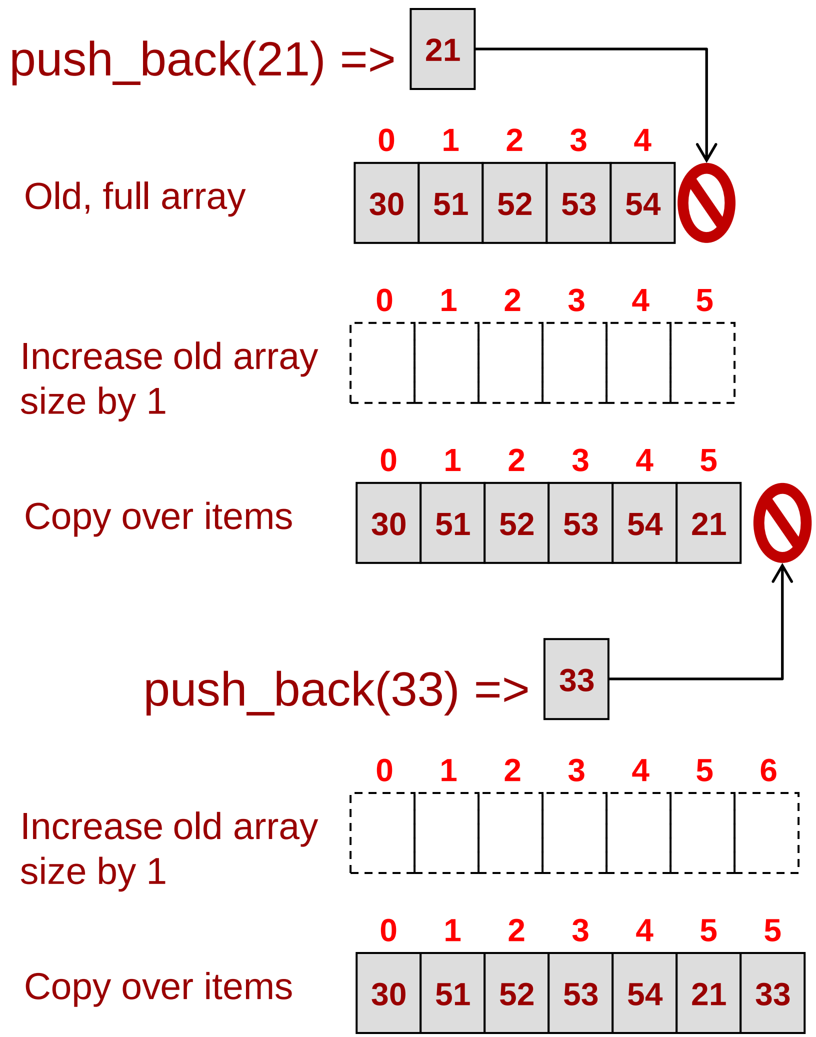
Amortized Run-time

- Used when it is impossible for the worst case of an operation to happen on each call (i.e. we can prove after paying a high cost that we will not have to pay that cost again for some number of future operations)
- Amortized Runtime = (Total runtime over k calls) / k
  - Average runtime over k calls
  - Use a "period" of calls from when the large cost is incurred until the next time the large cost will be incurred



Amortized Array Resize Run-time

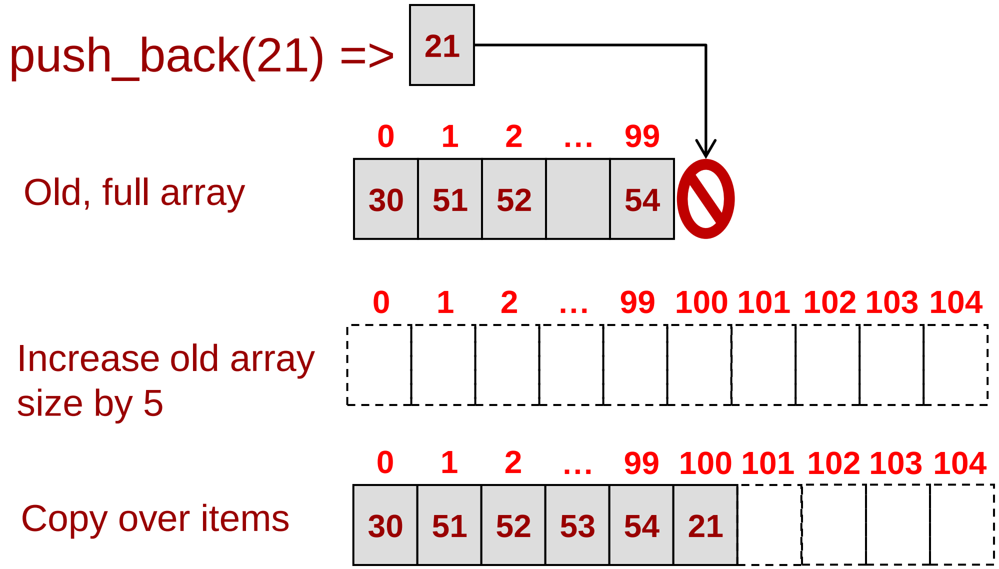
- What is the run-time of insert or push_back:
  - If we have to resize?
  - O(n)
  - If we don't have to resize?
  - O(1)
- Now compute the total cost of a series of insertions using resize by 1 at a time
- Each new insert costs O(n)… not good



Resize by 1 strategy

Amortized Array Resize Run-time

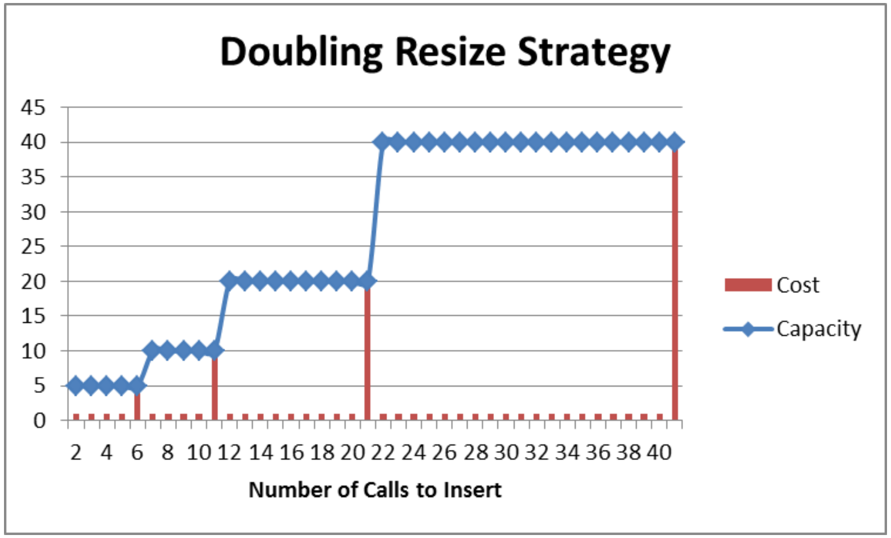
- What if we resize by adding 5 new locations each time
- Start analyzing when the list is full…
  - 1 call to insert will cost: n+1
  - What can I guarantee about the next 4 calls to insert?
    - They will cost 1 each because I have room
  - After those 4 calls the next insert will cost: (n+5)
  - Then 4 more at cost=1
- If the list is size n and full
  - Next insert cost = n+1
  - 4 inserts after than = 1 each = 4 total
  - Thus total cost for 5 inserts = n+5
  - Runtime = cost / inserts = (n+5)/5 = O(n)



Resize by 5 strategy

Consider a Doubling Size Strategy

- Start when the list is full and at size n
- Next insertion will cost?
  - O(n+1)
- How many future insertions will be guaranteed to be cost = 1?
  - n-1 insertions
  - At a cost of 1 each, I get n-1 total cost
- So for the n insertions my total cost was
  - n+1 + n-1 = 2*n
- Amortized runtime is then:
  - Cost / insertions
  - O(2*n / n) = O(2)
    = O(1) = constant!!!



Specialized List ADTs

STACKS AND QUEUE ADTS

Lists

- Ordered collection of items, which may contain duplicate values, usually accessed based on their position (index)
  - Ordered = Each item has an index and there is a front and back (start and end)
  - Duplicates allowed (i.e. in a list of integers, the value 0 could appear multiple times)
  - Accessed based on their position (list[0], list[1], etc.)
- What are some operations you perform on a list?

List Operations

Operation	Description	Input(s)	Output(s)
insert	Add a new value at a particular location shifting others back	Index : int Value	
remove	Remove value at the given location	Index : int	Value at location
get / at	Get value at given location	Index : int	Value at location
set	Changes the value at a given location	Index : int Value	
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		int
push_back / append	Add a new value to the end of the list	Value	
find	Return the location of a given value	Value	Int : Index

Stacks & Queues

- Lists are good for storing generic sequences of items, but they can be specialized to form other useful structures
- What if we had a List, but we restricted how insertion and removal were done?
  - **Stack** – Only ever insert/remove from one end of the list
  - **Queue** – Only ever insert at one end and remove from the other

First-In, First-Out (FIFOs)

QUEUE ADT

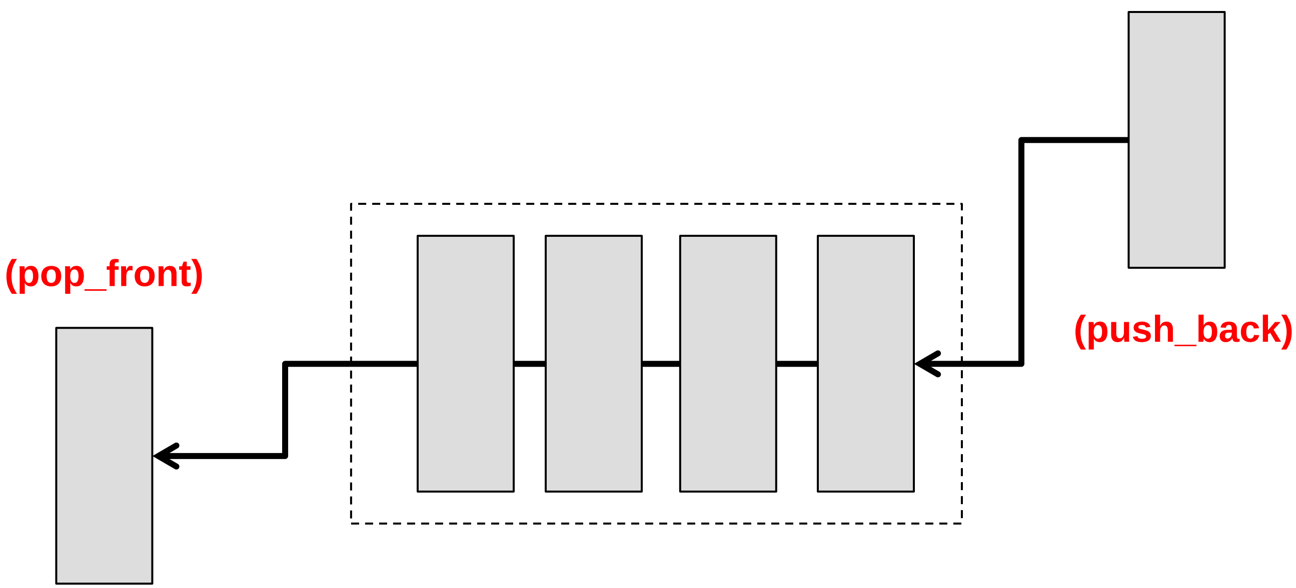
Queue ADT

- Queue – A list of items where insertion only occurs at the back of the list and removal only occurs at the front of the list
  - Like waiting in line for a cashier at a store
- Queues are FIFO (First In, First Out)
  - Items at the back of the queue are the newest
  - Items at the front of the queue are the oldest
  - Elements are processed in the order they arrive

A Queue Visual

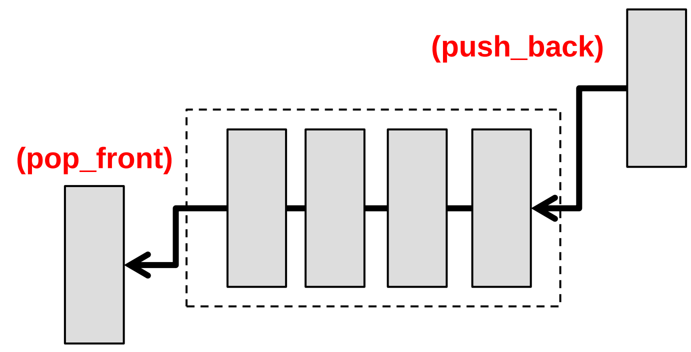
Items leave from the front (pop_front)

Items enter at the back (push_back)



Queue Operations

- What member functions does a Queue have?
  - push_back(item) – Add an item to the back of the Queue
  - pop_front() - Remove the front item from the Queue
  - front() - Get a reference to the front item of the Queue (don't remove it though!)
  - size() - Number of items in the Queue
  - empty() - Check if the Queue is empty



A Queue Class

- A sample class interface for a Queue
- Queue Error Conditions
  - **Queue Underflow** – The name for the condition where you call pop on an empty Queue
  - **Queue Overflow** – The name for the condition where you call push on a full Queue (a Queue that can't grow any more)
    - This is only possible for Queues that are backed by a bounded list

```
#ifndef QUEUEINT_H
#define QUEUEINT_H

class QueueInt  {
 public:
  QueueInt();
  ~QueueInt();
  size_t size() const;
  // enqueue
  void push_back(const int& value);
  // dequeue
  void pop_front();  // dequeue
  int const & front() const;
  bool empty() const;

 private:
  // ???
};
#endif
```

Other Queue Details

- How should you implement a Queue?
  - Compose using an ArrayList
  - Compose using a singly-linked list w/o a tail pointer
  - Compose using a singly-linked list w/ a tail pointer
  - Which is best?

	Push_back	**Pop_front**	**Front()**
ArrayList			
LinkedList (Singly-linked w/o tail ptr)			
LinkedList (Singly-linked w/ tail ptr)			

Queue Applications

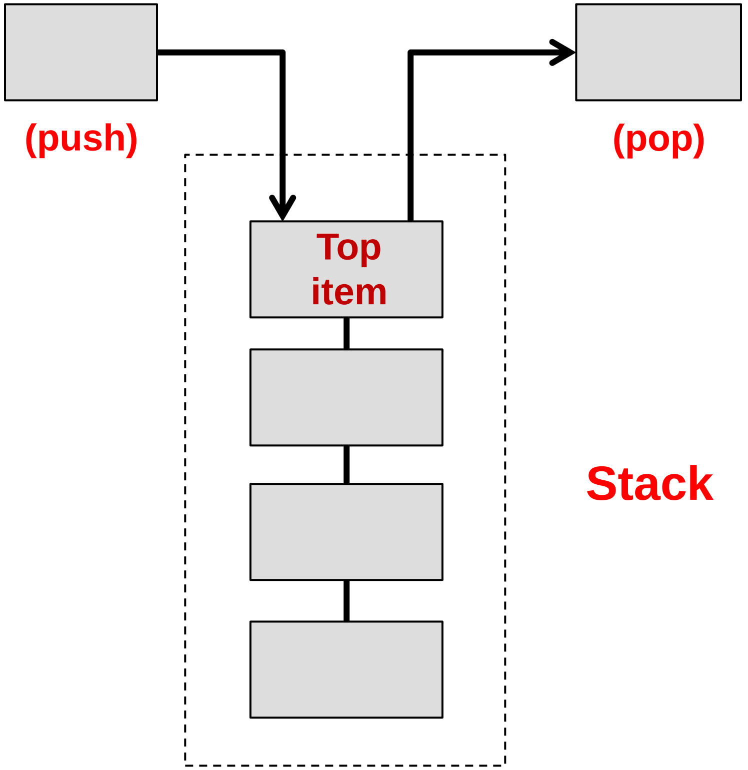
- Print Jobs
  - Click “Print” on the computer is much faster than actually printing (build a backlog)
  - Each job is processed in the order it's received (FIFO)
  - Why would you want a print queue rather than a print stack
- Seating customers at a restaurant
- Anything that involves "waiting in line"
- Helpful to decouple producers and consumers

Last-In, First-Out (LIFOs)

STACK ADT

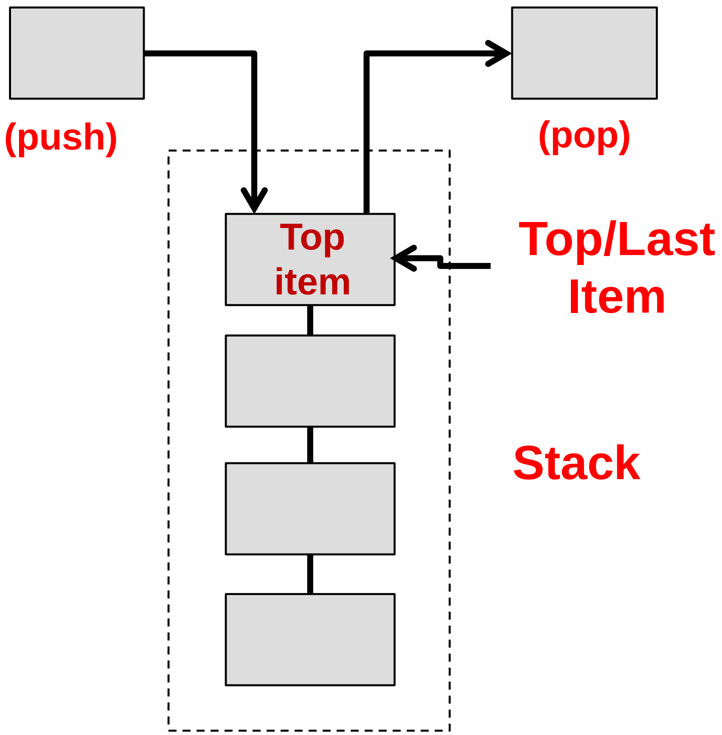
Stack ADT

- Stack: A list of items where insertion and removal only occurs at one end of the list
- Examples:
  - A stack of boxes where you have to move the top one to get to ones farther down
  - A spring-loaded plate dispenser at a buffet
  - A PEZ dispenser
  - Your e-mail inbox
- Stacks are LIFO
  - Newest item at top
  - Oldest item at bottom



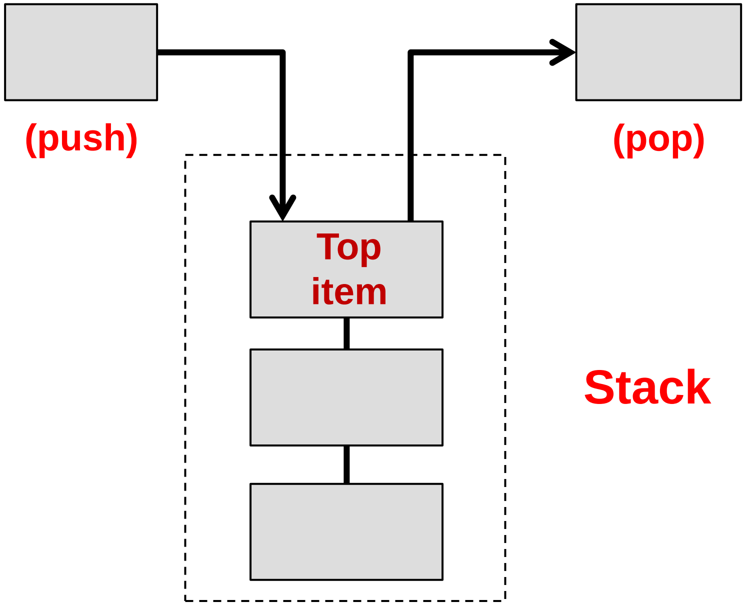
Stack Operations

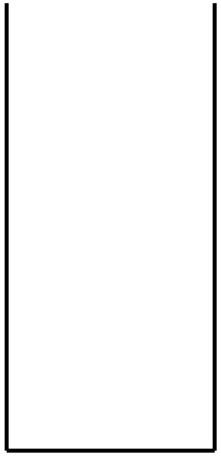
- What member functions does a Stack have?
  - push(item) – Add an item to the top of the Stack
  - pop() - Remove the top item from the Stack
  - top() - Get a reference to the top item on the Stack (don't remove it though!)
  - size() - Get the number of items in the Stack
- What member data does a Stack have?
  - A list of items
  - Top/Last Item Pointer/Index



Stack Axioms

- For all stacks, s:
  - `s.push(item).top() = item`
  - `s.push(item).pop() = s`
- Let's draw the stack for these operations:
  - `s.push(5).push(4).pop().top()`





A Stack Class

- A sample class interface for a Stack
- How should you implement a Stack?
  - Back it with an array
  - Back it with a linked list
  - Which is best?
- Stack Error Conditions
  - Stack Underflow – The name for the condition where you call pop on an empty Stack
  - Stack Overflow – The name for the condition where you call push on a full Stack (a stack that can't grow any more)

```
#ifndef STACKINT_H
#define STACKINT_H

class StackInt  {
 public:
  StackInt();
  ~StackInt();
  size_t size() const;
  bool empty() const;
  void push(const int& value);
  void pop();
  int const & top() const;
};
#endif
```

Array Based Stack

- A sample class interface for a Stack
- If using an array list, which end should you use as the "top"?
  - Front or back?
- If using a linked list, which end should you use?
  - If you just use a head pointer only?
  - If you have a head and tail pointer?

```
#ifndef STACKINT_H
#define STACKINT_H

class StackInt  {
 public:
  StackInt();
  ~StackInt();
  size_t size() const;
  bool empty() const;
  void push(const int& value);
  void pop();
  int const& top() const;
private:
  AListInt mylist_;
  // or LListInt mylist_;
};
#endif
```

Stack Examples

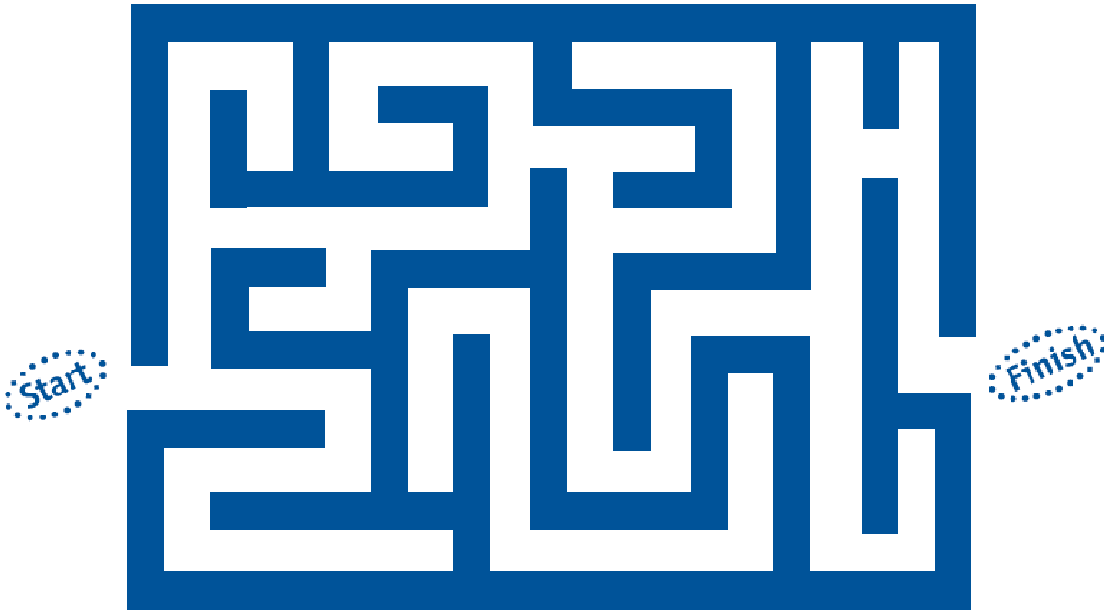
- Reverse a string

```
#include <iostream>
#include <string>
#include "stack.h"
using namespace std;
int main()
{
  StackChar s;

  string word;
  cout << "Enter a word: ";
  getline(cin,word);

  for(int i=0; i < word.size(); i++)
    s.push(word.at(i));

  while(!s.empty()){
    cout << s.top();
    s.pop();
  }
}
```

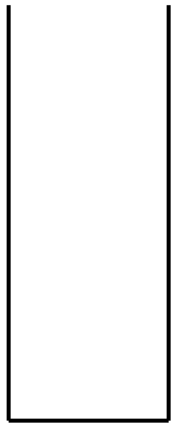
Type in: "hello"
Output: "olleh"

Another Stack Example

- Depth First Search (See Graph Traversals later in this semester)
- Use a stack whenever you encounter a decision, just pick and push decision onto stack. If you hit a dead end pop off last decision (retrace steps) and keep trying, etc.
  - Assume we always choose S, then L, then R
  - Strait or Left
    - Choose straight…dead end
    - Pop straight and make next choice…left
    - Next decision is Straight or Right…choose Straight…



http://www.pbs.org/wgbh/nova/einstein/images/lrk-maze.gif

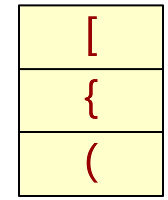


Stack Usage Example

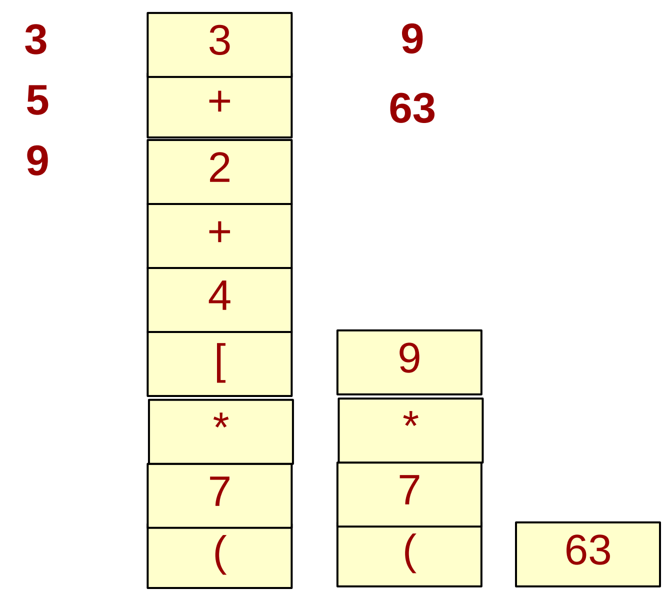
- Check whether an expression is properly parenthesized with '(', '[', '{', '}', ']', ')'
  - Correct: (7 * [8 + [9/{5-2}]])
  - Incorrect: (7*8
  - Incorrect: (7*8]
- Note: The last parentheses started should be the first one completed
- Approach
  - Scan character by character of the expression string
  - Each time you hit an open-paren: '(', '[', '{' **push** it on the stack
  - When you encounter a ')', ']', '}' the **top** character on the stack should be the matching opening paren type, otherwise ERROR!

(7 * { [8 + 9] / {5-2} })

({ [ ] { } })

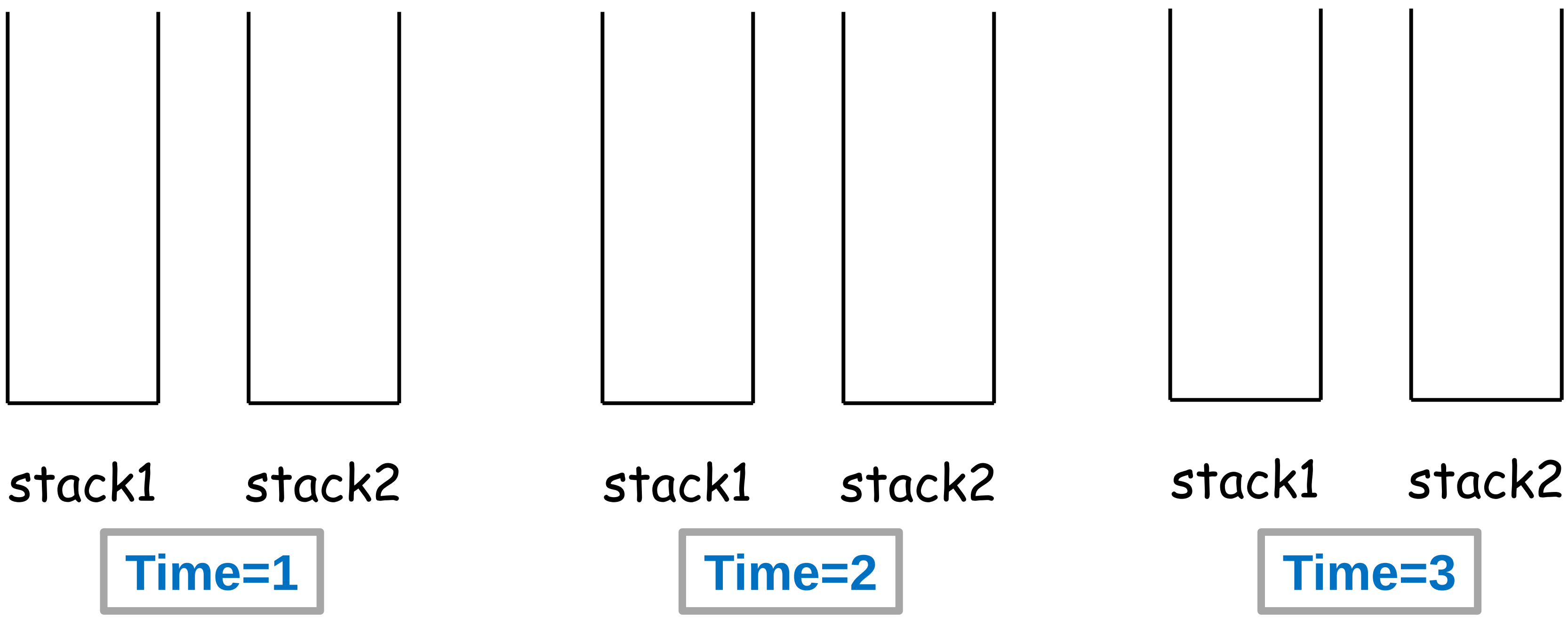


(7 * [4 + 2 + 3])



Queue with two stacks

- To enqueue(x), push x on stack 1
- To dequeue()
  - If stack 2 empty, pop everything from stack 1 and push onto stack 2.
  - Pop stack 2

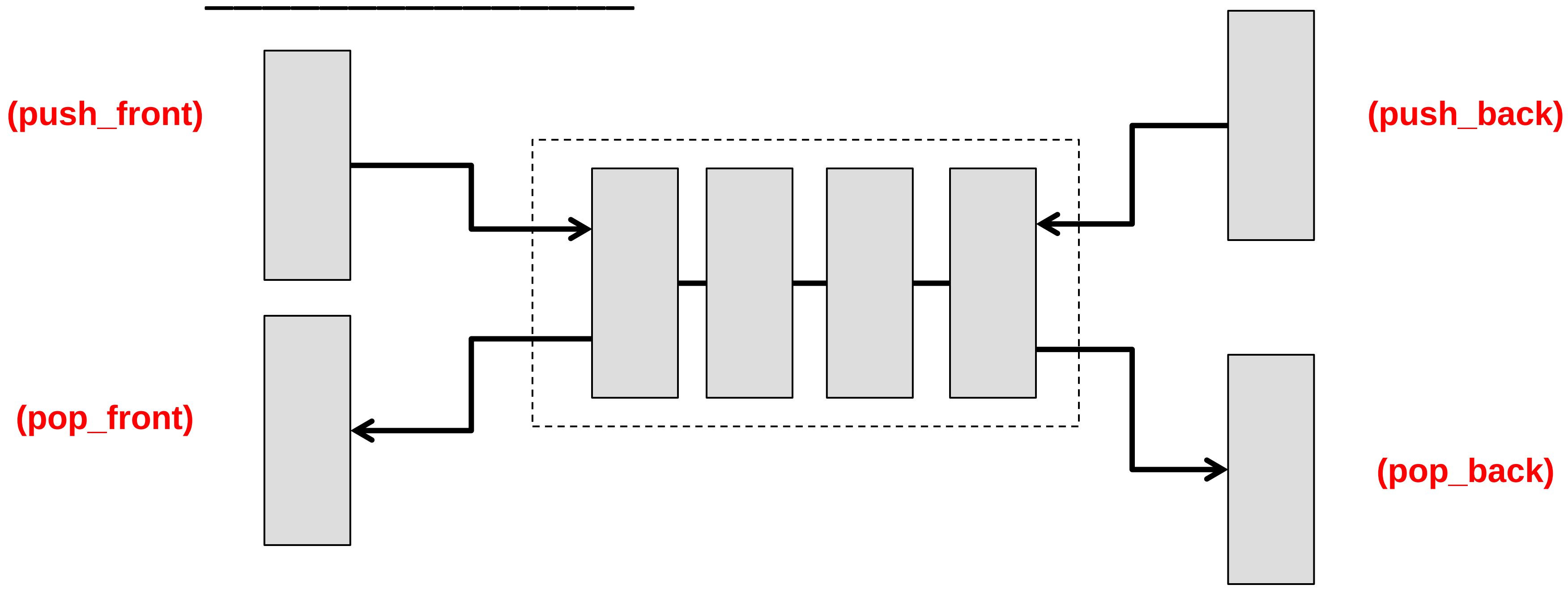


Double-ended Queues

DEQUE ADT

The Deque ADT

- **D**ouble-**e**nded **que**ues - Equally good (Θ(1)) push and pop on either end
- What list implementation supports this already?
  - ________________



STL Deque Class

- Uses an array-based approach
- Similar to vector but allows for push_front() and pop_front() options
- Useful when we want to put things in one end of the list and take them out of the other

1 my_deq

0	1	2	3	4
50	51	52	53	54

2 my_deq

0	1	2	3	4
51	52	53	54	60

after 1st iteration

3 my_deq

0	1	2	3	4
60	61	62	63	64

after all iterations

4 my_deq

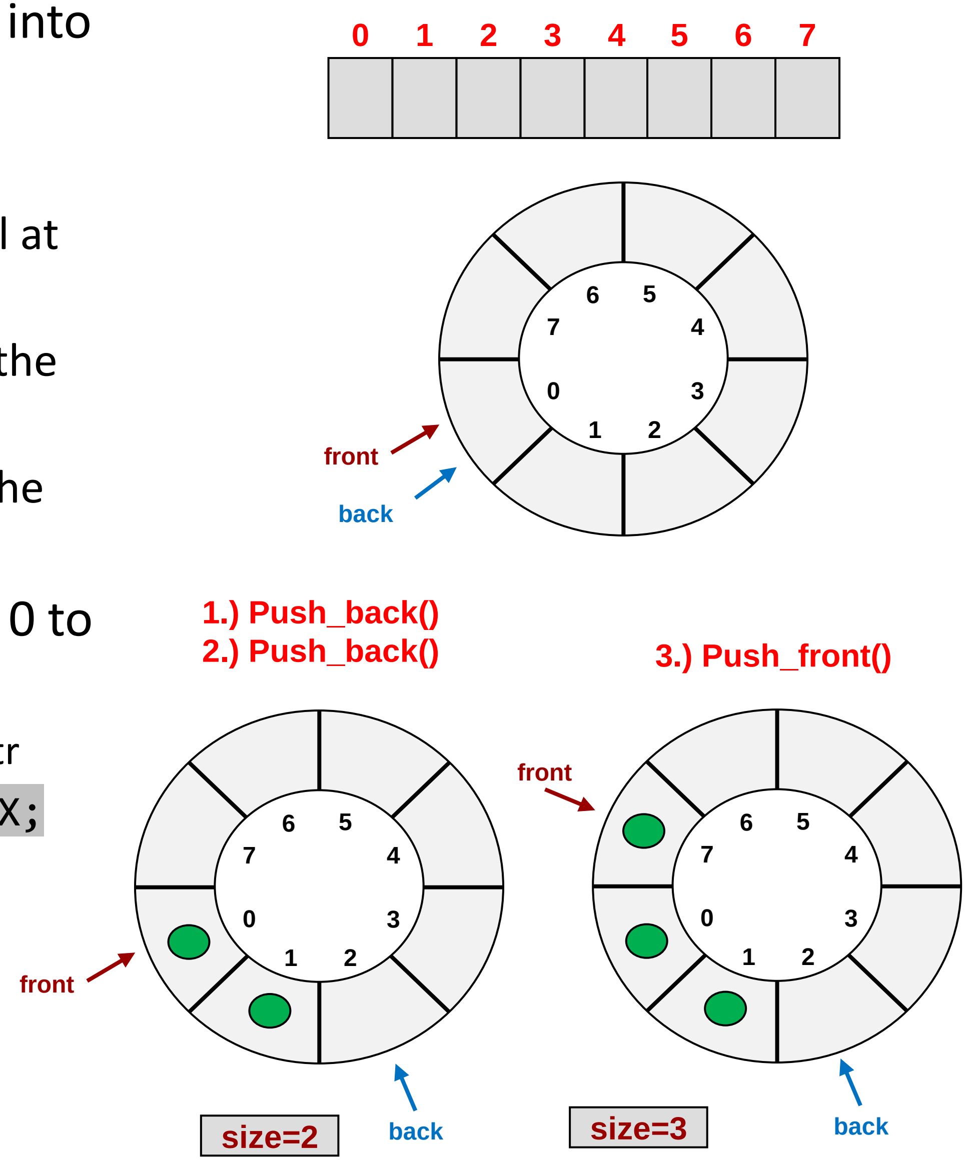
```
   #include <iostream>
   #include <deque>
   using namespace std;
   int main()
   {
     deque<int> my_deq;
     for(int i=0; i < 5; i++){
       my_deq.push_back(i+50);
     }
1    cout << "At index 2 is: " << my_deq[2] ;
     cout << endl;

     for(int i=0; i < 5; i++){
2      int x = my_deq.front();
       my_deq.push_back(x+10);
3      my_deq.pop_front();
     }
     while( ! my_deq.empty()){
       cout << my_deq.front() << " ";
       my_deq.pop_front();
4    }
     cout << endl;

   }
```

STL Vector vs. Deque

- std::vector is essentially a Dynamic Array List
  - Slow at removing and inserting at the front or middle
  - Fast at adding/remove from the back
  - Implies it could be used well as a (stack / queue)
- std::deque gives fast insertion and removal from front and back along with fast random access (i.e. get(i))
  - Almost has "look and feel" of linked list with head and tail pointers providing fast addition/removal from either end
  - Implies it could be used well as a (stack / queue)
  - Practically it is likely implemented as a circular array buffer

Circular Buffers

- Take an array but imagine it wrapping into a circle to implement a deque
- Setup a head and tail pointer
  - Head points at first occupied item, tail at first free location
  - Push_front() and pop_front() update the head pointer
  - Push_back() and pop_back() update the tail pointer
- To overcome discontinuity from index 0 to MAX-1, use modulo operation
  - Cannot just use `back++;` to move back ptr
  - Instead, use `back = (back + 1) % MAX;`
- Get item at index i
  - Must be relative to the `front` pointer



SOLUTIONS

Other Queue Details

- How should you implement a Queue?
  - Compose using an ArrayList
  - Compose using a singly-linked list w/o a tail pointer
  - Compose using a singly-linked list w/ a tail pointer
  - Which is best?

	Push_back	**Pop_front**	**Front()**
ArrayList	O(1)	O(n)	O(1)
LinkedList (Singly-linked w/o tail ptr)	O(n)	O(1)	O(1)
LinkedList (Singly-linked w/ tail ptr)	O(1)	O(1)	O(1)