Abstract Data Types

http://inst.eecs.berkeley.edu/~selfpace/studyguide/9C.sg/Output/ADTs.in.C.html

Data Structure Abstractions

- An abstract data type (ADT) is a data type (a set of values and a collection of operations on those values) that is accessed only through an interface.

- We refer to a program that uses an ADT as a client, and a program that specifies the data type as an implementation.

Why Use an ADT?

- Implement the functionality in different ways (memory use vs. speed) without changing client code.
  - Without even recompiling the client code (may have to re-link it though).

- Supports code re-use and modular programming.
  - Can limit the size and complexity for a given solution

- Easier to test localized functionality with driver

Linked List Abstraction

Using an array to represent a linked list

Linked List Abstraction

- A node could be laid out like this:

prev	DATA	next

- The list after a constructor call:

```
ArrayList <char> list(8);
//(head=-1, tail=-1)
```

-2		-2	-2		-2	-2		-2	-2		-2	-2		-2	-2		-2	-2		-2	-2		-2
	0			1			2			3			4			5			6			7	

Linked List Abstraction

```
template <typename T>
class ArrayList{
   private:
     struct ArrayNode{
        int prev_;
        T data_;
        int next_;
     };

    int head_;
    int tail_;
    unsigned capacity_;
    ArrayNode *list_;

   public:
     ArrayList(unsigned MaxElements);
     ~ArrayList();
     // ...
};
```

Linked List Abstraction

```
void TestArrayList(void){
  ArrayList <char> list(8);

  // A B C D E
  for (char c = 'A'; c < 'F'; ++c)
     list.push_back(c);

  list.insert('X', 0);      // X A B C D E
  list.remove_byindex(2);   // X A C D E
  list.remove_byindex(3);   // X A C E
  list.insert('Y', 2);      // X A Y C E
  list.push_front('P');     // P X A Y C E
  list.push_front('Q');     // Q P X A Y C E
  list.pop_back();          // Q P X A Y C
  list.pop_front();         // P X A Y C
  list[3] = 'J';            // P X A J C
}
```

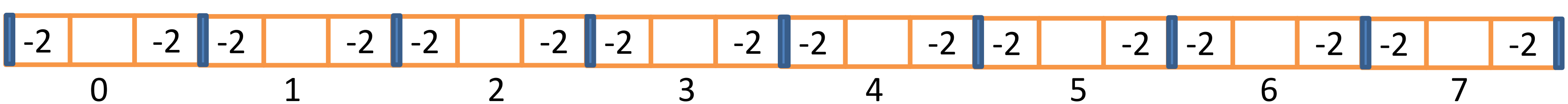
Different Implementations, SAME interface!

Linked List Abstraction

- A node could be laid out like this:

- The list after a constructor call:

```
ArrayList <char> list(8);
//(head=-1, tail=-1)
```



Linked List Abstraction

- Since 0 is a valid index, we need to choose another value to represent NULL.
- We also need a way to tell if a “node” is in use or not (EMPTY)

```
// 0 is a legal index, so -1 will be our NULL
const int NULL_NODE = -1;

// Set a node's prev/next to this to indicate it's availablity.
const int EMPTY_NODE = -2;

template <typename T>
ArrayList<T>::ArrayList(unsigned MaxElements)
{
    head_ = NULL_NODE;
    tail_ = NULL_NODE;
    capacity_ = MaxElements;
    list_ = new ArrayNode[capacity_];
}
```

Linked List Abstraction: Push_back

```
for (char c = 'A'; c < 'F'; ++c)
    list.push_back(c);
```

(head=0, tail=0)

-1	A	-1	-2		-2	-2		-2	-2		-2	-2		-2	-2		-2	-2		-2	-2		-2
	0			1			2			3			4			5			6			7	

(head=0, tail=1)

-1	A	1	0	B	-1	-2		-2	-2		-2	-2		-2	-2		-2	-2		-2	-2		-2
	0			1			2			3			4			5			6			7	

(head=0, tail=2)

-1	A	1	0	B	2	1	C	-1	-2		-2	-2		-2	-2		-2	-2		-2	-2		-2
	0			1			2			3			4			5			6			7	

(head=0, tail=3)

-1	A	1	0	B	2	1	C	3	2	D	-1	-2		-2	-2		-2	-2		-2	-2		-2
	0			1			2			3			4			5			6			7	

(head=0, tail=4)

-1	A	1	0	B	2	1	C	3	2	D	4	3	E	-1	-2		-2	-2		-2	-2		-2
	0			1			2			3			4			5			6			7	

Linked List Abstraction: Insertion

head=0, tail=4

-1	A	1	0	B	2	1	C	3	2	D	4	3	E	-1	-2		-2	-2		-2	-2		-2
	0			1			2			3			4			5			6			7	

```
list.insert('X', 0);
```

head=5, tail=4

5	A	1	0	B	2	1	C	3	2	D	4	3	E	-1	-1	X	0	-2		-2	-2		-2
	0			1			2			3			4			5			6			7	

Linked List Abstraction: Deletion

head=5, tail=4

0	1	2	3	4	5	6	7
5 A 1	0 B 2	1 C 3	2 D 4	3 E -1	-1 X 0	-2 -2	-2 -2

```
list.remove_byindex(2);
```

head=5, tail=4

0	1	2	3	4	5	6	7
5 A 2	-2 B -2	0 C 3	2 D 4	3 E -1	-1 X 0	-2 -2	-2 -2

```
list.remove_byindex(3);
```

head=5, tail=4

0	1	2	3	4	5	6	7
5 A 2	-2 B -2	0 C 4	-2 D -2	2 E -1	-1 X 0	-2 -2	-2 -2

Linked List Abstraction: Insertion

head=5, tail=4

0	1	2	3	4	5	6	7
5 A 2	-2 B -2	0 C 4	-2 D -2	2 E -1	-1 X 0	-2 -2	-2 -2

```
list.insert('Y', 2);
```

head=5, tail=4

0	1	2	3	4	5	6	7
5 A 1	0 Y 2	1 C 4	-2 D -2	2 E -1	-1 X 0	-2 -2	-2 -2

Linked List Abstraction: Push

head=5, tail=4

0			1			2			3			4			5			6			7		
5	A	1	0	Y	2	1	C	4	-2	D	-2	2	E	-1	-1	X	0	-2		-2	-2		-2

```
list.push_front('P');
```

head=3, tail=4

0			1			2			3			4			5			6			7		
5	A	1	0	Y	2	1	C	4	-1	P	5	2	E	-1	3	X	0	-2		-2	-2		-2

```
list.push_front(‘Q');
```

head=6, tail=4

0			1			2			3			4			5			6			7		
5	A	1	0	Y	2	1	C	4	6	P	5	2	E	-1	3	X	0	-1	Q	3	-2		-2

Linked List Abstraction: Pop

head=6, tail=4

5	A	1	0	Y	2	1	C	4	6	P	5	2	E	-1	3	X	0	-1	Q	3	-2		-2
	0			1			2			3			4			5			6			7	

`list.pop_back();`

Head=6, tail=2

5	A	1	0	Y	2	1	C	-1	6	P	5	-2	E	-2	3	X	0	-1	Q	3	-2		-2
	0			1			2			3			4			5			6			7	

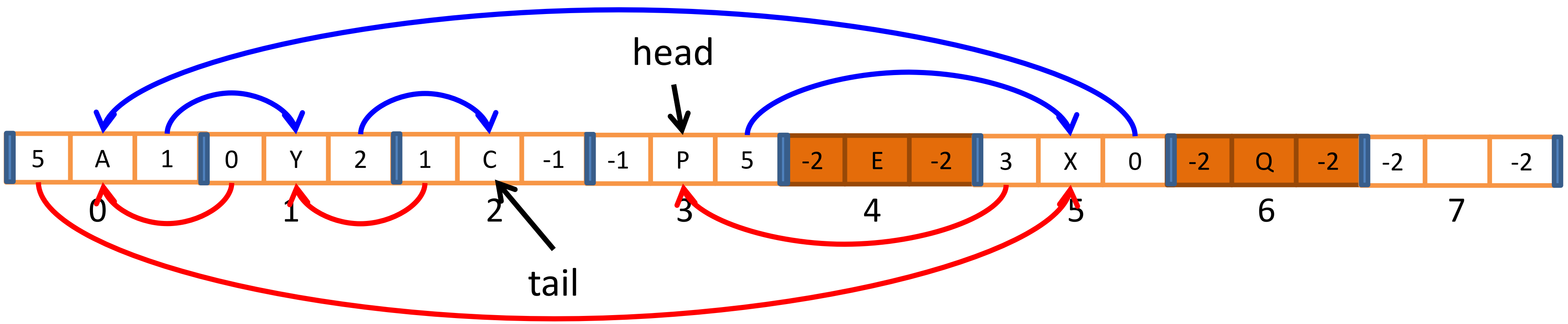
`list.pop_front();`

head=3, tail=2

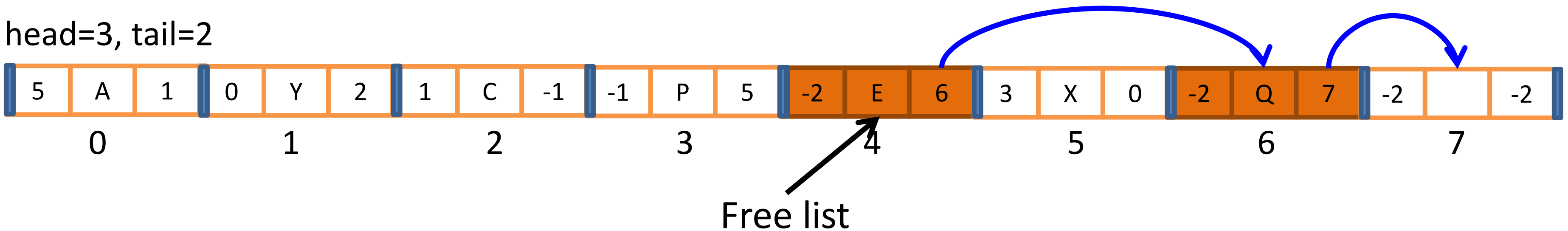
5	A	1	0	Y	2	1	C	-1	-1	P	5	-2	E	-2	3	X	0	-2	Q	-2	-2		-2
	0			1			2			3			4			5			6			7	

Linked List Abstraction

head=3, tail=2



Finding Empty Blocks?



Collections of ADTs

- Unlike a simple ADT (which is generally unique), collections have a common interface
  - Adding or Inserting an item
  - Removing an item
  - Counting the number of items
  - Searching the items (traversing)
  - Sorting the items
  - Performing some operation on all of the data

- Once we have the fundamental operations implemented, we can create specific ADTs (concrete types) from the more general ADT.

Useful Stack Example: Postfix Notation

Useful Stack Example: Postfix Notation

- Arithmetic expressions usually use infix notation: 3+4, 5*7+2

- Postfix notation has the operators after the operands: 3 4 +, 5 7 * 2 +

- Postfix has the nice property of not being ambiguous
  - You don't need parenthesis

How to evaluate an expression in postfix notation?

Evaluate

- A stack is the perfect data structure to implement this paradigm.

- Algorithm for evaluating expressions in postfix notation:
  1. When we see an operand, we push it on the stack.
  2. When we see an operator we:
     1. Perform the arithmetic: operand1 operator operand2, where operand1 and operand2 are popped from the stack.
     2. Push the result of the arithmetic into the stack
  3. When we have no more tokens, the answer is on the top of the stack (it will be the only item on the stack.)

Pushdown Stack ADT

Pushdown Stack

- Two basic operations:
  - Insert (push) a new item
  - remove (pop) the item that was most recently inserted.

- The stack is **LIFO** (last-in, first-out) paradigm.

- What data structure employs the **FIFO** (first-in, first-out) paradigm?

Interface

```
1. Stack(int capacity) //constructor

2. void Push(char item) //add an item to the top

3. char Pop(void)         //remove the top item

4. bool IsEmpty(void)   //check if empty
```

Stack Implementation #1 (Array)

```
class Stack1 {
   private:
     char *items;
     int size;
   public:
     Stack1(int capacity){
       items = new char [capacity];
       size = 0;
     }

     ~Stack1(){
       delete [] items;
     }

     void Push(char item){
        items[size++] = item;
     }

     char Pop(void){
        return items[--size];
     }

     bool IsEmpty(void){
        return (size == 0);
     }
};
```

```
int main(void){
   const int SIZE = 10;
   Stack1 stack(SIZE);

   const char *p = "ABCDEFG";
   for (unsigned i = 0; i < strlen(p); ++i)
      stack.Push(p[i]);

   while (!stack.IsEmpty())
      cout << stack.Pop();

   cout << endl;
   return 0;
}
```

Stack Implementation #1 (Array)

- (-) Only accepts **char** type
- (-) No error checking (Stack may be empty when calling Pop)
- (+) Complexity of push and pop?
- (-) If stack grows and shrinks, memory is wasted

Stack Implementation #2 (Linked-list)

```
class Stack2{
   private:
      CharItem *head;
      int size;
      int capacity;
   public:
     Stack2(int capacity){
        head = 0;
        this->capacity = capacity;
        size = 0;
     }

     ~Stack2(){
        while (head){
           CharItem *t = head->next;
           Free(head);
           head = t;
        }
     }

     void Push(char c){
        if (size >= capacity)
           return;

        CharItem *item = Allocate();
        item->data = c;
        item->next = head;
        head = item;
        ++size;
     }
```

```
     char Pop(void){
        char c = head->data;
        CharItem *temp = head;
        head = head->next;
        Free(temp);
        return c;
     }

     bool IsEmpty(void){
         return (head == 0);
     }
};
```

```
struct CharItem{
  CharItem *next;
  char data;
};

CharItem *Allocate(void){
  return new CharItem;
}

void Free(CharItem *item){
  delete item;
}
```

Stack Implementation #2 (Linked-list)

- (+) The client code doesn’t change at all. (Stack abstraction)
- (+) There is really no limit to the size.
- (+) The class includes capacity field to detect a full stack.
- (+) Complexity of push and pop?
- (+) Complexity of growing the stack? (no waste of memory space)
- (+) The class uses the generic Allocate and Free routines.
- (-) Still only accepts **char** type, no error checking/overhead.

Stack Implementation #2 (Linked-list)

```
struct CharItem{
  CharItem *next;
  char data;
};

CharItem *Allocate(void){
  return new CharItem;
}

void Free(CharItem *item){
  delete item;
}
```

We still have some limitations:

- Still only accepts **char** type.
- No error checking (e.g. Stack may be empty when calling Pop method.)
- With small data there can be significant overhead.
- However, there is no real size limit at all and we don't waste any space.

Stack Implementation #3 (Array)

```
class Stack1 {
   private:
     char *items;
     int size;
   public:
     Stack1(int capacity){
       items = new char [capacity];
       size = 0;
     }

     ~Stack1(){
       delete [] items;
     }

     void Push(char item){
        items[size++] = item;
     }

     char Pop(void){
        return items[--size];
     }

     bool IsEmpty(void){
        return (size == 0);
     }
};
```

```
template <typename Item>
class Stack3 {
   private:
     Item *items;
     int size;
   public:
     Stack3(int capacity){
       items = new Item [capacity];
       size = 0;
     }

     ~Stack3(){
       delete [] items;
     }

     void Push(Item item){
        items[size++] = item;
     }

     Item Pop(void){
        return items[--size];
     }

     bool IsEmpty(void){
        return (size == 0);
     }
};
```

Stack Implementation #3 (Array)

```
int main(void){
   const int SIZE = 10;
   //Stack1 stack(SIZE);
   Stack3 <char> stack(SIZE);

   char *p = "ABCDEFG";
   for (unsigned i = 0; i < strlen(p); ++i)
      stack.Push(p[i]);

   while (!stack.IsEmpty())
      cout << stack.Pop();

   cout << endl;
   return 0;
}
```

```
The output: (char)
GFEDCBA
```

```
int main(void){
   const int SIZE = 5;
   Stack3 <int> stack(SIZE);

   for (unsigned i = 1; i <= SIZE; ++i){
     cout << 1000 * i << endl;
     stack.Push(1000 * i);
   }
   cout << endl;

   while (!stack.IsEmpty())
      cout << stack.Pop() << endl;
   return 0;
}
```

```
The output: (int)

1000
2000
3000
4000
5000

5000
4000
3000
2000
1000
```

Stack Implementation #3 (Array)

- (+) The client code changes are minimal (but only once and it's simple)

- (-) It's still an array, so we have pros/cons of that data type (complexity of growth).

- (+) Accepts almost any data type.

Stack Implementation #4 (Linked-list)

- Using linked-lists of generic pointers.
- We use this Node structure:

```
struct Item{
    Item *next;
    void *data;
};
```

- Similar to the second version, change CharItem to Item (generic)
- The data is untyped (`void *`)
- Simpler than a **template** class (not C++ specific) with possibly less memory requirements, but not as safe.

Stack Implementation #4 (Linked-list)

```
class Stack4{
   private:
      Item *head;
      int size;
      int capacity;
   public:
      Stack4(int capacity){
         head = 0;
         size = 0;
         this->capacity = capacity;
      }

      ~Stack4(){
         //walk the list and delete each item
         while (head){
            Item *t = head->next;
            Free(head);
            head = t;
         }
      }

      void Push(void *data){
         if (size >= capacity) //stack is full
            return;            //do nothing
         Item *item = Allocate(); //allocate item

         item->data = data; // insert at head
         item->next = head;
         head = item;
         ++size;
      }

      void *Pop(void){
         void *p = head->data; //get top item
         Item *temp = head;    //update head
         head = head->next;
         Free(temp);       //deallocate
         return p;
      }

      bool IsEmpty(void){
         return (head == 0);
      }
};


int main(void){
      const int SIZE = 10;
      Stack4 stack(SIZE);

      char *p = "ABCDEFG";

      for (unsigned i = 0; i < strlen(p); ++i)
         stack.Push(&p[i]); //push address of data;

      while (!stack.IsEmpty()){
            char *c = (char *) stack.Pop();
            cout << *c; // dereference data;
      }

      cout << endl;
}
```

Stack Implementation #4 (Linked-list)

```
struct TStudent{
   int ID;
   float GPA;
   int Year;
     };

int main(void){
   const int SIZE = 5;
   Stack4 stack(SIZE);

   for (int i = 0; i < SIZE; ++i){
      TStudent *ps = new TStudent;
      ps->GPA = GetRandom(100, 400) / 100.0;
      ps->ID = GetRandom(1, 1000);
      ps->Year = GetRandom(1, 4);
      cout << "Student ID: " << ps->ID << ", Year: " << ps->Year << ", GPA: " << ps->GPA << endl;
      stack.Push(ps);
   }
   cout << endl;
   while (!stack.IsEmpty()){
      TStudent *ps = (TStudent *) stack.Pop();
      cout << "Student ID: " << ps->ID << ", Year: " << ps->Year << ", GPA: " << ps->GPA << endl;
   }
   return 0;
     }
```

Stack Implementation #4 (Linked-list)

```
The output:


Student ID: 468, Year: 3, GPA: 1.41
Student ID: 170, Year: 1, GPA: 1.12
Student ID: 359, Year: 3, GPA: 1.4
Student ID: 706, Year: 2, GPA: 1.83
Student ID: 828, Year: 2, GPA: 2.04


Student ID: 828, Year: 2, GPA: 2.04
Student ID: 706, Year: 2, GPA: 1.83
Student ID: 359, Year: 3, GPA: 1.4
Student ID: 170, Year: 1, GPA: 1.12
Student ID: 468, Year: 3, GPA: 1.41
```

- The implementation is simple and will deal with any pointer.
- The memory usage of the stack is independent of the size of the data (always `sizeof(void*)`).
- This class is not as type-safe as a template class.
- The client will always interact in the same way, that is pushing addresses.

Useful Stack Example: Postfix Notation

Useful Stack Example: Postfix Notation

- Arithmetic expressions usually use infix notation: 3+4, 5*7+2
- Postfix notation has the operators after the operands: 3 4 +, 5 7 * 2 +
- Postfix has the nice property of not being ambiguous
  - You don’t need parenthesis

Examples

- Infix notation, with parenthesis
  - 5 * (((9 + 8) * (4 + 6)) + 7) = 2075
- Infix notation, without parenthesis
  - 5 * 9 + 8 * 4 * 6 + 7 = 244
- Postfix notation
  - 5 9 8 + 4 6 + * 7 + * = 2075
- Infix notation, with parenthesis
  - 5 * 9 + (8 * 4) * (6 + 7) = 461
- Postfix notation
  - 5 9 * 8 4 * 6 7 + * + = 461

How to evaluate an expression in postfix notation?

Let’s CODE it!!

- A stack is the perfect data structure to implement this paradigm.

- Algorithm for evaluating expressions in postfix notation:
    1. When we see an operand, we push it on the stack.
    2. When we see an operator we:
        1. Perform the arithmetic: operand1 operator operand2, where operand1 and operand2 are popped from the stack.
        2. Push the result of the arithmetic into the stack
    3. When we have no more tokens, the answer is on the top of the stack (it will be the only item on the stack.)

Let’s CODE it!!

```
int Evaluate_Postfix(const char *postfix){
   Stack <int> stack(strlen(postfix));

   while(*postfix){
     char token = *postfix;

     if(token == '+')
        stack.Push(stack.Pop() + stack.Pop());
     else if(token == '*')
        stack.Push(stack.Pop() * stack.Pop());
     else if (token >= '0' && token <= '9')
        stack.Push(token - '0');
     ++postfix;
   }
   return stack.Pop();
}
```

```
void main (void)
{
  char postfix [256];

  cout << "Enter the operations" << endl;
  cin.width(256);
  cin >> postfix;
  cout << postfix << " = " <<
  Evaluate_Postfix(postfix) << endl;
}
```

```
598+46**7+* = 2075

34+ = 7

34+7* = 49

12*3*4*5*6* = 720
```

```
int Evaluate_Postfix(const char * postfix)
{
	Stack<int> stack(strlen(postfix));

	while(*postfix)
	{
		char token = *postfix;

		if(token == '+')
			stack.Push(stack.Pop() + stack.Pop());
		else if(token == '*')
			stack.Push(stack.Pop() * stack.Pop());
		else if (token >= '0' && token <= '9')
			stack.Push(token - '0');

		postfix++;
	}
	return stack.Pop();
}
```

```
void main (void)
{
  char postfix [256];

  cout << "Enter the operations" << endl;
  cin.width(256);
  cin >> postfix;
  cout << postfix << " = " << Evaluate_Postfix(postfix) << endl;
}
```

```
598+46**7+* = 2075

34+ = 7

34+7* = 49

12*3*4*5*6* = 720
```

Exercise

- Modify the Evaluate function above to support subtraction and division as well.

- Note: You'll need to pay attention to the order of operands.

- Try it with `(2 * 8 / 4 + 5 * 6 - 8)` which is `(2 8 * 4 / 5 6 * + 8 -)` in postfix.

Converting Infix to postfix

Converting Infix to postfix

- Input: An infix expression
- Output: A postfix expression

- Examples
  - 5 * (((9 + 8) * (4 * 6)) + 7) = 2075
    =>5 9 8 + 4 6 * * 7 + * = 2075
  - 5 * 9 + (8 * 4) * (6 + 7) = 461
    =>5 9 * 8 4 * 6 7 + * + = 461
  - 2 * 5 * 2 * 8 + 4 + 5 + 3 = 172
    =>2 5 2 8 * * * 4 5 3 + + + = 172

Converting Infix to postfix

1. Operand: send to the output
2. Left parenthesis – push onto the stack
3. Right parenthesis – operators are popped off the stack and sent to the output until a left parenthesis is found (and then discarded).
4. Operator
    1. If the stack is empty, push the operator
    2. If the top of the stack is a left parenthesis, push the operator onto the stack.
    3. If the top of the stack is an operator which has the same or lower precedence than the scanned operator, push the scanned operator.
    4. If the top of the stack is an operator which has a higher precedence, pop the stack and send to the output. Repeat the algorithm, with the new top of stack.

5. If the input stream is empty and there are still operators on the stack, pop all of tem and add them to the output.

Note: the only symbols that exist on the stack are operators and left parentheses. Operands and right parentheses are never pushed onto the stack.

Try All These!

- 3+4, 5*7+2
- 5 * (((9 + 8) * (4 + 6)) + 7)
- 5 * 9 + (8 * 4) * (6 + 7) = 461
- 2 * 5 * 2 * 8 + 4 + 5 + 3 = 172

Stack Implementations: Array v.s. Linked-list

Array v.s. Linked-list

```
class Stack1 {
   private:
     char *items;
     int size;
   public:
     Stack1(int capacity){
       items = new char [capacity];
       size = 0;
     }

     ~Stack1(){
       delete [] items;
     }

     void Push(char item){
        items[size++] = item;
     }

     char Pop(void){
        return items[--size];
     }

     bool IsEmpty(void){
        return (size == 0);
     }
};
```

```
class Stack2{
   private:
      CharItem *head;
      int size;
      int capacity;
   public:
     Stack2(int capacity){
        head = 0;
        this->capacity = capacity;
        size = 0;
     }

     ~Stack2(){
        while (head){
           CharItem *t = head->next;
           Free(head);
           head = t;
        }
     }

     void Push(char c){
        if (size >= capacity)
           return;

        CharItem *item = Allocate();
        item->data = c;
        item->next = head;
        head = item;
        ++size;
     }

     char Pop(void){
        char c = head->data;
        CharItem *temp = head;
        head = head->next;
        Free(temp);
        return c;
     }

     bool IsEmpty(void){
         return (head == 0);
     }
};
```

Array v.s. Linked-List

- Scalability?
  - For a large amount of data, a list is the better choice since an **array re-size** is an expensive operation.
  - With a list, you don't need to allocate any more space than you actually need so it's more space efficient.

- The locality of the linked list is not as good as the locality of the array.

Queue

Queue

- We usually mean a **FIFO** queue (First-in, First-Out)
  - Add an item to the front.
  - Remove from the back.

```
int main(){
  Queue<char> q;

  q.Push('A');
  q.Push('B');
  q.Push('C');

  cout << q.Pop();
  cout << q.Pop();
  cout << q.Pop();
}
```

Implementation?

- Linked-list?
  - Expensive to add to the end: $\mathrm{O(n)}$.
  - Use a tail pointer and double-linked list for removing from the end.

- Array?
  - Expensive to remove from the front: $\mathrm{O(n)}$.
  - Use a circular array.

- Implementing a FIFO Queue as a linked list is trivial.
- Implementing it using an array (efficiently) is slightly more interesting.

FIFO Queue Using a Circular Array

Setup

- We will use an array of `SIZE` elements.

- We have to keep track of the start and end of the array.
  - `if (tail == head) the queue is empty.`
  - `if ((tail+1)%SIZE==head), the queue is full.`
  - Number of items in queue is `(tail – head + SIZE) % SIZE.`

- We keep one unused slot to distinguish between full and empty

- A circular array gives us $O(1)$ for both adding and removing.

Example

```
Queue queue(5);
queue.Add(7);
queue.Add(4);
queue.Add(5);
queue.Remove();
queue.Add(6);
queue.Add(8);
queue.Add(2);
```

```
if (tail == head)
    the queue is empty.
if ((tail+1)%SIZE==head)
    the queue is full.
```

Number of items in queue is (tail – head + SIZE) % SIZE.

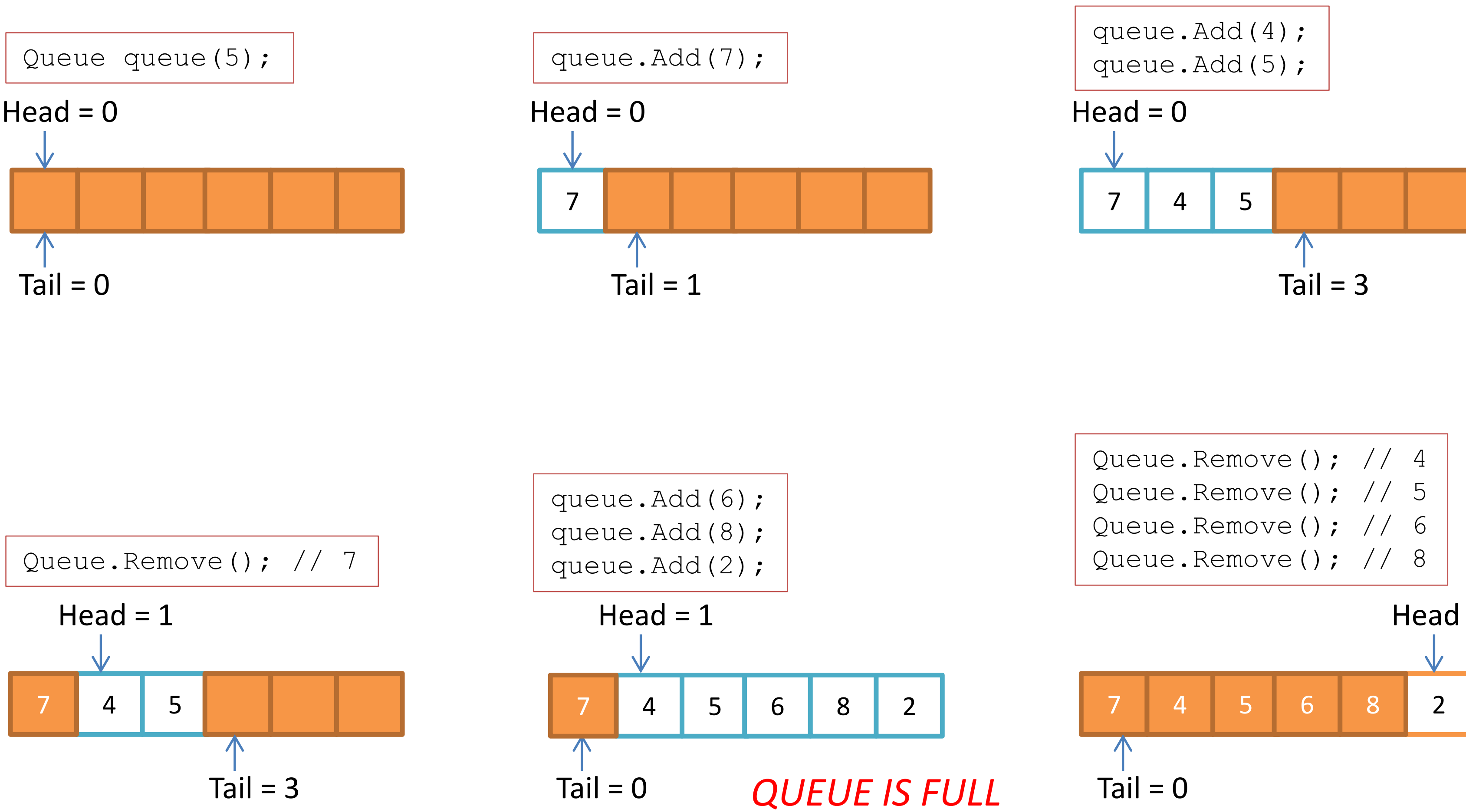
Example

```
for(int i=1;i<=4;++i)
  queue.Remove(); // 2

for(int i=1;i<=4;++i)
  queue.Add(i); // 2

for(int i=1;i<=6;++i)
  queue.Remove(); // 2
```

```
if (tail == head)
    the queue is empty.
if ((tail+1)%SIZE==head)
    the queue is full.
```

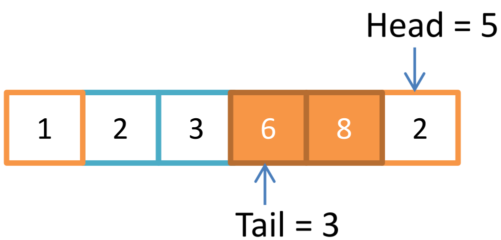
Number of items in queue is (tail – head + SIZE) % SIZE.

Example

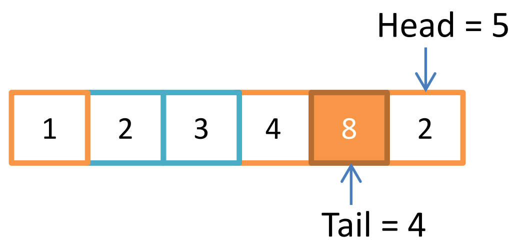

Queue queue(5);
Head = 0
Tail = 0
queue.Add(7);
Head = 0
7
Tail = 1
queue.Add(4);
queue.Add(5);
Head = 0
7
4
5
Tail = 3
Queue.Remove(); // 7
Head = 1
7
4
5
Tail = 3
queue.Add(6);
queue.Add(8);
queue.Add(2);
Head = 1
7
4
5
6
8
2
Tail = 0
QUEUE IS FULL
Queue.Remove(); // 4
Queue.Remove(); // 5
Queue.Remove(); // 6
Queue.Remove(); // 8
Head = 5
7
4
5
6
8
2
Tail = 0

Example

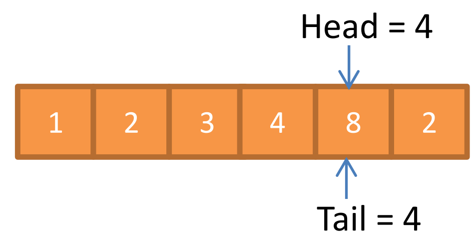
```
queue.Add(1);
queue.Add(2);
queue.Add(3);
```




```
queue.Add(4);
```




```
Queue.Remove(); // 2
Queue.Remove(); // 1
Queue.Remove(); // 2
Queue.Remove(); // 3
Queue.Remove(); // 4
Queue.Remove(); // empty!
```



Exercise

- Using this class interface, Implement the Queue as a circular array.

```
class Queue{
   public:
      Queue(int MaxItems);
      ~Queue();
      void Add(int Item); // Push
      int Remove(void);   // Pop
      bool IsFull(void) const;
      bool IsEmpty(void) const;
};
```

Considerations

- Both `Queues` and `Stacks` can be implemented using arrays or linked-lists.

- The interface of `Queue` is essentially the same as that of `Stack`. (Many implementations use the names Push and Pop.)

- Depending on how you implemented them changes the complexity to $\mathrm{O}(\mathrm{n})$ to $\mathrm{O}(1)$.

Priority Queue

Priority Queue

- Pushes to the front, pops the item with the highest priority.
  - Priority is user-defined
    - Could be the item with the highest id.
    - Could be the item with the oldest timestamp (think of messaging systems).

- Can be implemented using an array or a linked-list.

Priority Queue

```
class PriorityQueue{
   private:
   // private data

   public:
      PriorityQueue(int capacity);
      ~PriorityQueue(void);
      void Add(int Item);
      int Remove(void);
      bool IsEmpty(void) const;
      bool IsFull(void) const;
      void Dump(void) const;
};
```

Abstract Interface

Priority Queue

```
class PQArray{
   private:
      int *array_;
      int capacity_;
      int count_;
   public:
   // public interface
};
```

```
struct PQNode{
   PQNode *next;
   int data;
};


class PQList{
   private:
      PQNode *list_;
      int capacity_;
      int count_;
   public:
   // public interface
};
```

Priority Queue

- Complexity depends on how the list/array is implemented
  - Sorted?
  - Unsorted?

- Which of the implementations has a more efficient Add method?

- Which of the implementations has a more efficient Remove Method?

Example

```
int main(void){
    PQList pq(10);
    // Sorted linked list implementation

    pq.Add(4);  pq.Add(7);  pq.Add(2);
    pq.Add(5);  pq.Add(8);  pq.Add(1);

    pq.Dump();

    printf("Removing: %i\n", pq.Remove());
    pq.Dump();

    printf("Removing: %i\n", pq.Remove());
    pq.Dump();

    printf("Removing: %i\n", pq.Remove());
    pq.Dump();

   return 0;
}
```

<u>And the associated output:</u>

```
8 7 5 4 2 1
```

Removing: 8

```
7 5 4 2 1
```

Removing: 7

```
5 4 2 1
```

Removing: 5

```
4 2 1
```

Example

```
int main(void){
    PQArray pq(10);
    // Unsorted array implementation

    pq.Add(4);  pq.Add(7);  pq.Add(2);
    pq.Add(5);  pq.Add(8);  pq.Add(1);

    pq.Dump();

    printf("Removing: %i\n", pq.Remove());
    pq.Dump();

    printf("Removing: %i\n", pq.Remove());
    pq.Dump();

    printf("Removing: %i\n", pq.Remove());
    pq.Dump();

    return 0;
}
```

And the associated output:

```
4 7 2 5 8 1
Removing: 8
4 7 2 5 1
Removing: 7
4 1 2 5
Removing: 5
4 1 2
```

Considerations

- Result is the same.
- Implementation are different.
- Complexities are different.

- Exercise: Using the class interface above, implement two priority queues. One using an array and one using a linked list. You can decide whether or not to keep it sorted.

Stack, Queue, Priority Queue

- Example: postfix expression evaluation
- 3 6 2 + 5 4 + 2 4 * * + -
- Stack:
- Queue:
- Priority Queue (assume smaller number is of higher priority)