Stacks and Queues

In this lecture:
- Variants on Lists
- Stacks
- Queues

Variants of Lists

- Previous lecture: double ended list (constant time access to front and end of list). Simpler variant: only access to the head of the list is constant time, to get to the tail need to traverse the list.
- Circular lists (ring buffers)
- Doubly linked lists

Circular lists

- Last link points back to first link.
- Terminating traversals of the list:
- Maintain separate count of list size, or:
- Sentinel element at end of list

- Usage: circular buffers

Doubly linked lists

- Each link points to its successor *and* predecessor in the list.

- Useful when the list should be traversible in both directions. Example of use: text processor where every line is an element in the list.

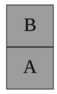
Stacks and Queues

- Not so much for data storage: more for organising programs

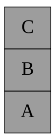
Stack: Last In, First Out

Stack : Last In, First Out

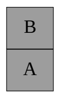
Stack : Last In, First Out



Stack : Last In, First Out



Stack : Last In, First Out

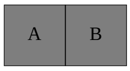


Stack : Last In, First Out

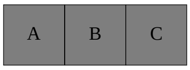
Queue: First In, First Out

Queue: First In, First Out

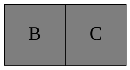
Queue: First In, First Out



Queue: First In, First Out



Queue: First In, First Out



Queue ADT

As for lists, need an ADT for items which are kept there. Let us call it **`ItemType`** (in Java, could be Object).

Methods

- **`void enqueue(ItemType item)`**

  Postcondition: item added to end of queue.
- **`boolean isempty()`**

  Postcondition: returns true if queue is empty

Methods contd.

- **`ItemType dequeue()`**

Precondition: queue is not empty

Postcondition: returns item at front of queue and deletes it from the queue.

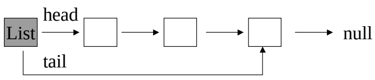
- **`ItemType peek()`**

Precondition: queue is not empty

Postcondition: returns item at front of queue.

List implementation of queue

- Uses double ended list like the one from the previous lecture.
- enqueue(Object item): insert at tail
- dequeue(Object item): remove and return the head of the list.

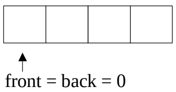


Array implementation of queue

- Two indices: one for the front and one for the back of the queue.
- Deleting from front: increment front index
- Adding to back: increment back index.
- The queue wraps around, but front and back are not allowed to cross past each other.

Array implementation of Queue

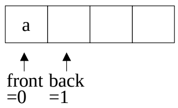
Example:



size=0
capacity=4

Array implementation of Queue

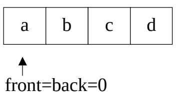
Enqueue an item:



size=1
capacity=4

Array implementation contd.

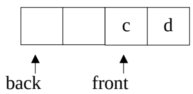
- Make the queue full:



size=capacity=4

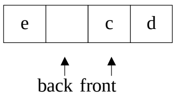
Array implementation contd.

- Delete some things:



Array implementation contd.

- Insert more things:



Java for the array implementation

- Fields: **array arr; int front; int back; int size; int capacity**

```
• isempty(){
     return (size == 0);
  }
• peek() { if !isempty()
       return arr[front];
           else // throw exception}
```

Array implementation contd.

```
• enqueue(Object x){
if (size == capacity) throw an
  exception: queue is full
else {
   arr[back] = x;
   size++;
   if (back==capacity-1) back = 0;
   else back++;
} }
```

Array implementation contd.

```
• dequeue(){
if (size == 0) throw an exception:
  queue is empty
else {
   value = arr[front];
   if(front==capacity-1) front=0;
   else front++;
   size--;
   return value;
}}
```

Stack ADT

- Logical domain: stacks of **ItemType**.
- Methods:
  - **boolean isempty()**

    Postcondition: returns true if stack is empty
  - **ItemType peek()**

    Precondition: stack is not empty

    Postcondition: item at top of stack returned.

Stack Methods contd.

- **void push (ItemType item)**

Postcondition: item added to top of stack

- **ItemType pop()**

Postcondition: item at top of stack returned and deleted from stack

Variations

- If we consider stacks of fixed maximal size, need the following variation:
- **`boolean full()`**

Postcondition: returns true if stack is full

- **`void push(ItemType item)`**

Precondition: stack is not full.

Postcondition: item added to top of stack.

Stack Implementations

- Vector implementation:
- Makes more sense when the size of stack is known in advance.
- No dynamic memory allocation or deallocation required once stack is created (unless we have to do dynamic re-sizing).
- Constant time access to top of stack

Stack implementations

- Linked list implementation:
- No limits on size of stack, and size need not be known in advance.
- Dynamic memory allocation and deallocation whenever items are popped onto or pushed off the stack
- Constant time access to top of stack.

Linked List Implementation

- Use a simple linked list
- **`peek() { return the value at head;}`**
- **`push() {insert an item at the head;}`**
- **`pop() {return the value at head and reset the head to be the head.next link;}`**

Vector Implementation

- Could have done an array implementation, but resizable array (vector) really makes more sense.
- Fields: resizable vector **`vec`** holding the stack and index of stack top (first free position in the stack), **`top`**. Stack grows to the right.

Vector Implementation

- Methods:

```
• boolean isempty() {
     return (top == 0) ;
  }
• Object peek() {
     return vec.elementAt(top-1);
}// will throw an exception if the
  stack is empty
```

Vector Implementation

```
• void push(Object item) {
     vec.add(top++, item);
}
• Object pop() {
     if (isempty) // throw exception
     else {
       return vec.elementAt(--top);
     }
}
```

Summary

- Stacks and queues are very important ADTs and have many uses.
- Stacks: LIFO (last in, first out).
- Queues: FIFO (first in first out).
- Can be implemented in different ways.
- For stacks, vector implementation is better if maximum size of stack can be predicted.
- Queues: array/vector or linked list implementation (list implementation is more straightforward).

Eliminating Recursion

- Recursion overheads
- Tail recursion elimination
- Elimination of recursion using a stack (don't try this at home)

Recursion is Expensive

- Recursive algorithms are elegant and usually easier to understand and implement, but recursive calls are expensive.
- Each recursive call is usually implemented by placing all the necessary information (return address, parameters, local variables) onto a stack. Each return pops the corresponding record of the stack.
- This uses both time (in setting up the record) and space on the stack.

Recursive factorial

```
recFactorial(n) {  // assume n >=0
  if (n <= 1) {
    return 1;
  } else {
    return n*recFactorial(n-1);
  }
}
```

Iterative Factorial

```
itFactorial(n) {
   result = 1;
   while (n > 1) {
        result = result * n;
        n--;
  }
return result;
}
```

Tail recursion optimisation

- Even if you don't replace recursion by iteration in this case, it is likely that a decent compiler will.
- Instead of keeping all intermediate values on the stack, it will update the same variable (using constant space)
- In some cases this is not possible and explicit stack is used: does not save a lot of space but is still more efficient.

Just for illustration: factorial again

```
stackFactorial(n) {
   Stack stack = newStack();
   while (n>1) stack.push(n--);
   result = 1;
   while(! stack.isempty()) {
      result = result * stack.pop();
   }
   return result;
}
```

Real Example: Quicksort

```
public void recQuickSort(int[] arr,
  int left, int right){
   if (right - left) <= 0) return;
   else {
    int border = partition(arr, int
  left, int right);
    recQuickSort(arr, left, border-
  1);
    recQuickSort(arr, border+1,
  right);
   }
}
```

Quicksort using a stack

```
public void stackQuickSort(int[]
  arr, int left, int right){
   Stack stack = new Stack(); //
   int border;
   // assume the stack can keep ints
   stack.push(left);
   stack.push(right);
```

Iterative Quicksort contd.

```
while(! stack.isempty()){
      int j = stack.pop();
      int i = stack.pop();
      border = partition(arr, i, j);
```

Iterative Quicksort contd.

```
if((border-1) - i > 1) {
         stack.push(i);
         stack.push(border-1);
      }
      if(j - (border+1) > 1) {
         stack.push(border+1);
         stack.push(j);
      }
   }
}
```

Example

Consider sorting the following array:

0	1	2	3	4
17	3	8	16	5

Stack: 4
0

Example

After the first iteration:

0	1	2	3	4
5	3	8	16	17

Stack: 4
3
1
0

Example

After the second iteration:

0	1	2	3	4
5	3	8	16	17

Stack: 1
0

Example

After the third iteration:

0	1	2	3	4
3	5	8	16	17

Stack:

Further reading

- Shaffer, chapter 4 (4.2 and 4.3).

Summary

- Recursion can always be eliminated using an explicit stack if you need a more efficient algorithm.
- However the resulting implementation maybe more difficult to understand and debug.