CS122 Algorithms and Data Structures

MW 11:00 am - 12:15 pm, MSEC 101

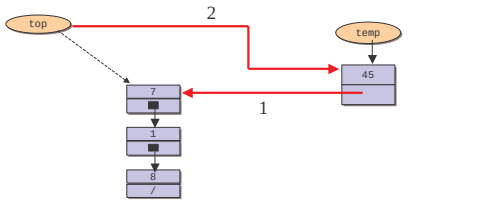
Instructor: Xiao Qin

Lecture 6: Stacks and Queues

Announcements

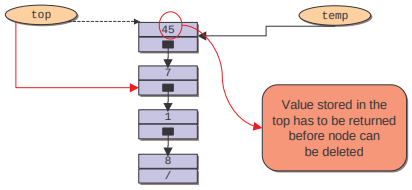
- Quiz 1 results
- Homework 2 is available
- Due on September 29th, 2004
- www.cs.nmt.edu/~xqin/courses/cs122

push(45)



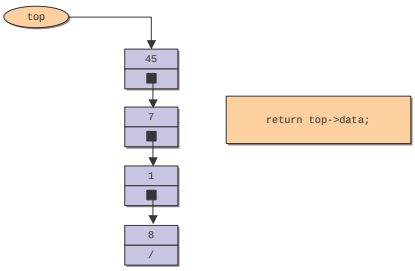

```
NodeType *temp = new NodeType;
temp->data = n;
temp->next = top;
top = temp;
temp = NULL;
```

pop()




```
NodeType *temp = top;
top = top->next;
int returnValue = temp->data;
delete temp;
return returnValue;
```

top()




```
return top->data;
```

Applications of Stacks

- Examining programs to see if symbols balance properly. (Bracket matching)
- Performing “postfix” calculations.
- Performing “infix” to “postfix” conversions.
- Function calls (and especially recursion) rely upon stacks.

Symbol Balancing

- A useful tool for checking your code is to see if the (), {}, and [] symbols balance properly.
  - For example the sequence …[…(…)…]… is legal but the sequence …[…(…]…)… is not.
  - The presence of one misplaced symbol can result in hundreds of worthless compiler diagnostic errors!

Symbol Balancing Algorithm

- The algorithm is simple and efficient:
  - Make an empty stack.
  - Read characters until end of file (EOF).
  - If the character is an opening symbol ([{, push it onto the stack.
  - If it is a closing symbol)]}, then if the stack is empty report an error. Otherwise pop the stack.
  - If the symbol popped is not the corresponding opening symbol, report an error.
  - At EOF, if the stack is not empty report an error.

Example

```
class StackNode
{
private:
  int element;
  StackNode *next;

public:
  StackNode (const int &theElement,
             StackNode *n = NULL)
    : element(theElement),
      next(n)
  { }
}
```

Push the { on the stack.

Push the (on the stack.
See the) and pop the (. Everything OK.
Push the (, see the), and pop the (. OK.
Push the (, see the), and pop the (. OK.
Push the {, see the }, and pop the {. OK.
See the }, pop the original {. All OK.

Postfix Calculator

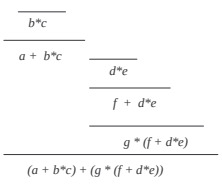
- What is “postfix”?
  - It is the most efficient method for representing arithmetic expressions.
  - With “infix” (the method you are used to) you put operators between operands: *a* + *b*.
  - With “postfix” you put operators after operands: *a b* +.
  - There is never any need to use ()’s with postfix notation. There is never any ambiguity.

Postfix Example

- This example is in infix:

  a + *b* * *c* + (*d* * *e* + *f*) * *g*
- In postfix this is:

  a b c * + *d e* * *f* + *g* * +



Postfix Calculator Algorithm

- The algorithm is simple and efficient O(*N*):
  - Read in input.
  - If input is an operand, push on stack.
  - If input is an operator, pop top two operands off stack, perform operation, and place result on stack.
- Example: *a b c* * +
  - Push *a*, *b*, and then *c* on the stack.
  - Pop *c* and *b*, perform multiplication, and push result.
  - Pop (*b***c*) and *a*, perform addition, and push result.

Infix to Postfix Conversion

- Surprisingly, one only needs a stack to write an algorithm to convert an infix expression to a postfix expression.
  - It can handle the presence of ()'s.
  - It is efficient: O(*N*).

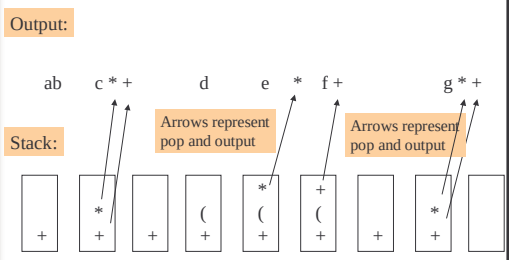
Conversion Algorithm

- The algorithm is reasonably simple:
  - Read infix expression as input.
  - If input is operand, output the operand.
  - If input is an operator +, -, *, /, then pop and output all operators of >= precedence. Push operator.
  - If input is (, then push.
  - If input is), then pop and output all operators until see a (on the stack. Pop the (without output.
  - If no more input then pop and output all operators on stack.

Conversion Example

- Infix*:*
  *a + b * c + (d * e + f) * g*
- In postfix this is:
  *a b c * + d e * f + g * +*
- Try to follow the algorithm to obtain the right postfix expression.

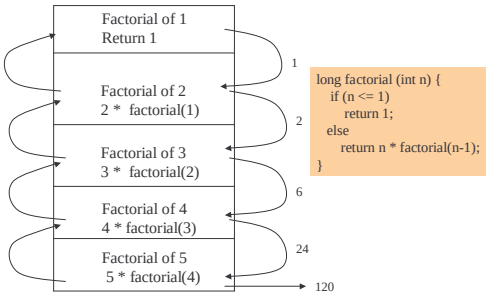
Details of Example



Function Calls

- In almost all programming languages, calling a function involves the use of a stack.
  - When there is a function call, all of the important information (values of variables etc.) is stored on a stack, and control is transferred to the new function.
  - This is especially important during recursion.

Example



Stack Overflow Problems

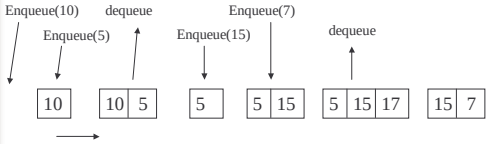
- The factorial program is an example of "tail recursion", where the recursive call occurs at the last line of the program.
  - Probably exceed the stack limit in your operating system
  - Probably exceed the capacity of your integers or even doubles.

Queues

- A queue is a waiting line
- Both ends are used: one for adding elements and one for removing them.
- FIFO structure: First in, First out

Queues

A series of operations executed on a queue



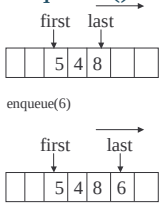
Queue ADT

- Data:
  - A collection of homogeneous elements arranged in a sequence. Elements are inserted to the end and removed from the front
- Operations:
  - Enqueue
  - Dequeue
  - IsEmpty
  - firstEl

Implementation Details

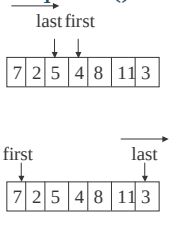
- Contiguous memory
  - Using a circular array
- Linked lists
  - Using a node structure to store data and a pointer to the next node: a chain of nodes

Queue Implementation via Arrays enqueue()




```
if (!ifFull()) {
    if (last==size-1||last ==-1) {
        storage[0]= el;
        last = 0;
        if (first == -1)
            first = 0;
    }
    else {
        last++;
        storage[last]= el;
    }
}
```

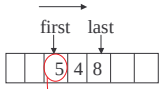
Queue Implementation via Arrays enqueue()




```
if (!ifFull()) {
    if (last==size-1||last ==-1) {
        storage[0]= el;
        last = 0;
        if (first == -1)
            first = 0;
    }
    else {
        last++;
        storage[last]= el;
    }
}
```

```
first == last + 1
      OR
first == 0 and last == size - 1;
```

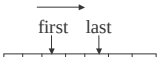
Queue Implementation via Arrays dequeue()



Value stored in the tmp has to be returned before first is updated

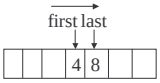
```
tmp = storage[first];
if (first = last)
    last = first = -1;
else if (first = size –1)
        first = 0;
     else first++;
return tmp
```

Queue Implementation via Arrays dequeue()



dequeue() Queue is empty

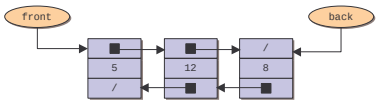
```
tmp = storage[first];
if (first = last)
    last = first = -1;
else if (first = size –1)
        first = 0;
     else first++;
return tmp
```



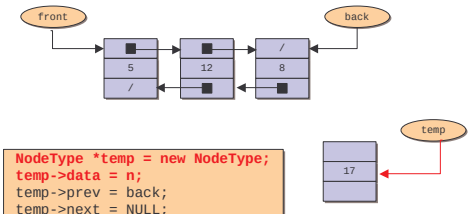
Queues using Linked Lists

- Implementation of queues using linked lists resembles the implementation of doubly linked lists with some operations being restricted
- Being a queue we have only one insert operation called `enqueue()`.
  - In many ways push is the same as insert in the back
- We have also one delete operation called `dequeue()`
  - This operation is the same as the operation delete from the front
- The other important operations in a stack, called `front()` and `isEmpty()`, don't modify the structure

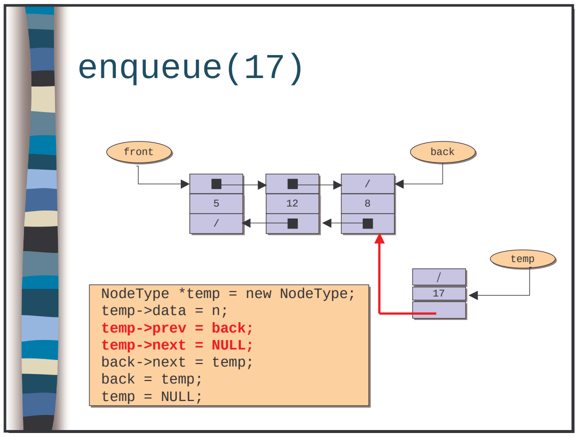
Pictorial view of a queue

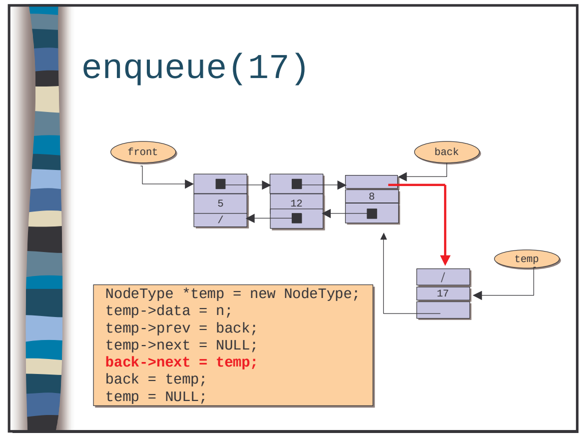


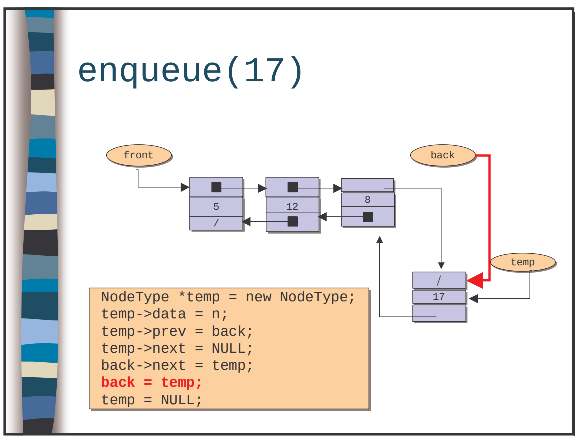
enqueue(17)

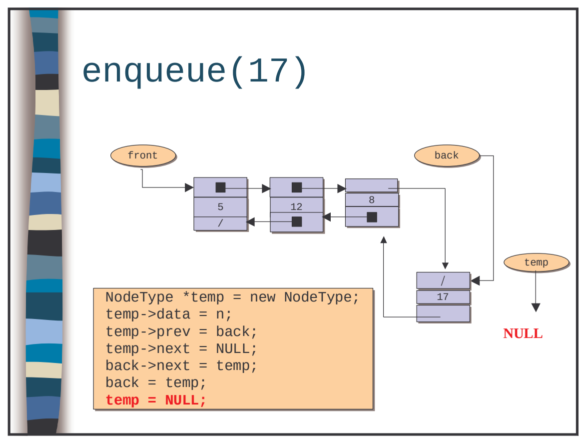


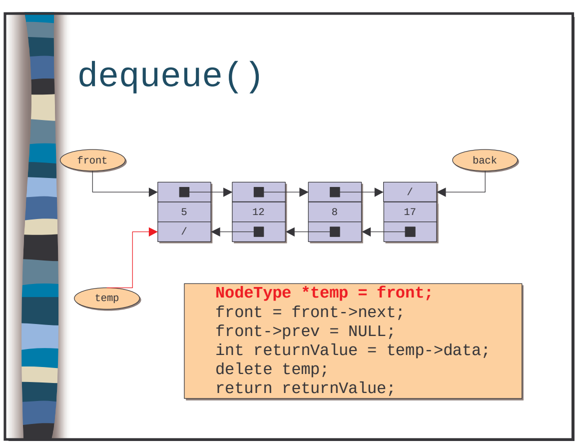

```
NodeType *temp = new NodeType;
temp->data = n;
temp->prev = back;
temp->next = NULL;
back->next = temp;
back = temp;
temp = NULL;
```

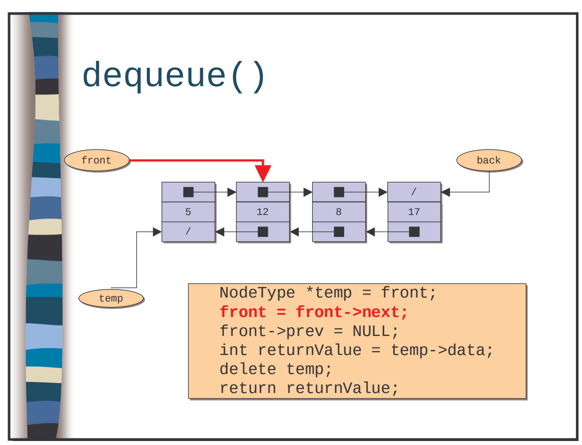

enqueue(17)
front
back
temp
5
12
8
17
NodeType *temp = new NodeType;
temp->data = n;
temp->prev = back;
temp->next = NULL;
back->next = temp;
back = temp;
temp = NULL;


enqueue(17)
front
back
temp
5
12
8
17
NodeType *temp = new NodeType;
temp->data = n;
temp->prev = back;
temp->next = NULL;
back->next = temp;
back = temp;
temp = NULL;

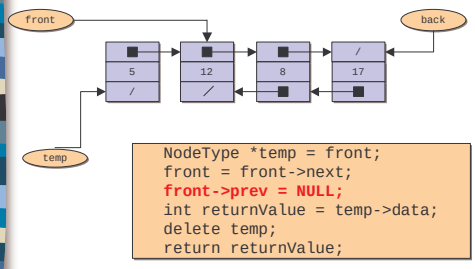

enqueue(17)
front
back
temp
5
12
8
17
NodeType *temp = new NodeType;
temp->data = n;
temp->prev = back;
temp->next = NULL;
back->next = temp;
back = temp;
temp = NULL;


enqueue(17)
front
back
temp
NULL
5
12
8
17
NodeType *temp = new NodeType;
temp->data = n;
temp->prev = back;
temp->next = NULL;
back->next = temp;
back = temp;
temp = NULL;


dequeue()
front
back
temp
5
12
8
17
NodeType *temp = front;
front = front->next;
front->prev = NULL;
int returnValue = temp->data;
delete temp;
return returnValue;


dequeue()
front
back
temp
5
12
8
17
NodeType *temp = front;
front = front->next;
front->prev = NULL;
int returnValue = temp->data;
delete temp;
return returnValue;

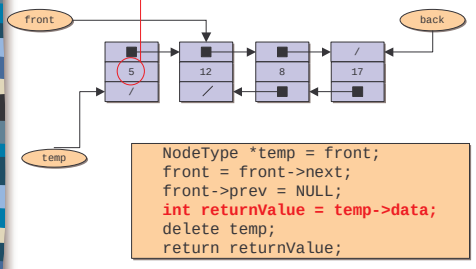
dequeue()




```
NodeType *temp = front;
front = front->next;
front->prev = NULL;
int returnValue = temp->data;
delete temp;
return returnValue;
```

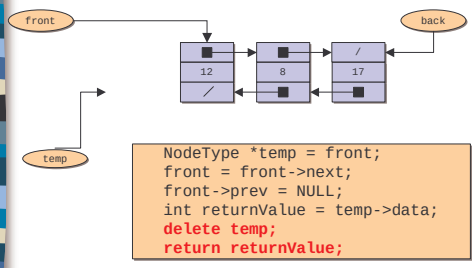
dequeue()

The value has to be "saved" to a temporary variable before the node can be deleted




```
NodeType *temp = front;
front = front->next;
front->prev = NULL;
int returnValue = temp->data;
delete temp;
return returnValue;
```

dequeue()




```
NodeType *temp = front;
front = front->next;
front->prev = NULL;
int returnValue = temp->data;
delete temp;
return returnValue;
```