Computer Science

Dr. Y. Liow (June 6, 2024)

Contents

106 Linked list **5100**

106.1 ADT: Abstract data type debug: adt.tex . . . 5106
106.2 Singly linked list debug: singlylinkedlist.tex . . . 5107
106.2.1 Singly linked node class . . . 5110
106.2.2 Insert head . . . 5115
106.2.3 Insert tail . . . 5119
106.2.4 Delete head . . . 5120
106.2.5 Delete tail . . . 5122
106.2.6 Find . . . 5123
106.2.7 Exercises . . . 5124
106.2.8 Singly linked list class . . . 5129
106.2.9 Iterators . . . 5134
106.3 Sentinel node debug: sentinel-node.tex . . . 5137
106.4 Doubly linked list debug: doublylinkedlist.tex . . . 5140
106.4.1 Insertion . . . 5141
106.4.2 Deletion . . . 5145
106.4.3 Destructor, copy constructor, assignment operator . . . 5146
106.5 Circular linked list debug: circularlinkedlist.tex . . . 5151
106.6 Stack debug: stack.tex . . . 5152
106.7 Queue debug: queue.tex . . . 5155
106.8 Double–ended queue debug: deque.tex . . . 5157
106.9 C++ STL singly linked list: `std::forward_list` debug: cpp-stl-forward-list.tex . . . 5158
106.10 C++ STL doubly linked list: `std::list` debug: cpp-stl-list.tex . . . 5162
106.11 C++ STL deque: `std::deque` debug: cpp-stl-deque.tex . . . 5166
106.12 C++ STL stack: `std::stack` debug: cpp-stl-stack.tex . . . 5167
106.13 C++ STL queue: `std::queue` debug: cpp-stl-queue.tex . . . 5169
106.14 Applications debug: mathematical-expressions.tex . . . 5171
106.15 Reverse debug: reverse.tex . . . 5172
106.16 Palindromes debug: palindromes.tex . . . 5174
106.17 Balanced expressions debug: balanced-expressions.tex . . . 5175
106.18 Polish notation debug: polish-notation.tex . . . 5178
106.19 Postfix notation; reverse Polish notation (RPN) debug: rpn.tex . . . 5183

106.20 Infix notation debug: infix.tex 5186
106.21 Infix to RPN debug: infix-to-rpn.tex 5201
106.22 Stack machine debug: stack-machine.tex 5207
106.23 Long integer debug: long-integer.tex 5209
106.24 Sparse long integer debug: sparse-long-integer.tex 5210
106.25 Sparse matrix debug: sparse-matrix.tex 5212
106.26 Graphs: Adjacency matrix and adjacency list debug: adjacency-list.tex 5214
106.27 More exercises debug: exercises.tex 5219

Chapter 106

Linked list

Solutions

Solution to Exercise 106.27.1.

debug: exercises/exercises0/answer.tex

Solution not provided.

Solution to Exercise 106.27.2.

debug: exercises/exercises1/answer.tex

Solution not provided.

Solution to Exercise 106.27.3.

debug: exercises/exercises2/answer.tex

Solution not provided.

Solution to Exercise 106.27.4.

debug: exercises/exercises3/answer.tex

Solution not provided.

Solution to Exercise 106.27.5.

debug: exercises/exercises4/answer.tex

Solution not provided.

106.1 ADT: Abstract data type debug: adt.tex

abstract data type

An **abstract data type** (ADT) is just a fancy name for a type. The description of the type includes a description of the operations on objects/values of that type without actually describing the implementation.

Think about a battery. If I say I have an "AAA" battery, then I'm saying the battery has a certain standard size and you can use the battery in a device that expects an AAA battery. The type "AAA" does not tell you the chemicals are used in that battery, i.e., "AAA" does not specify the "implementation" of this battery. (But if I tell you the brand and the model of this AAA battery, then that would reveal the implementation.)

An example of an ADT is the stack. The stack ADT is a type for containers with mainly two operations:

- `push`: for putting a value into the stack
- `pop`: for removing a value from the stack

Therefore other auxiliary operations such as `is_empty` that tells you if the stack is empty or not. The `push` and `pop` works this way: If the stack is not empty, pop will remove from the stack the value that was last pushed onto the stack. The stack is an example of a **self-organizing container** because you don't tell the stack where to put the value that is inserted into the stack. You can (and should) think of a stack as a stack of plates at a buffet: the plate you take from the stack of plates is the plate on top of the stack which is the last plate placed on top of the stack. The value that is popped from the stack is called the **top** of the stack. This is the reason why the stack is also called a **LIFO**, **last-in-first-out**, data structure. Here, a **data structure** refers to a container of values with a particular organizational behavior.

self-organizing container

top

LIFO
last-in-first-out
data structure

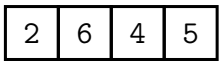
In terms of implementation, you can for instance use an array with a size variable (or `std::vector`) where the top of the stack is the value at the last index position. Based on this implementation you can then talk about the runtimes of the push and pop operation.

Another implementation is if you again use an array with a size variable (or vector), but the top of the stack correspond to the value at index 0. This stack implementation would have a different runtime from the one above.

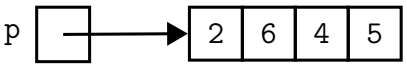
106.2 Singly linked list debug: singlylinkedlist.tex

Recall that you have seen the first type of container: arrays (static or dynamic). The problem with the array (both static and dynamic) is the that memory required by an array is continguous. When the memory is sufficiently fragmented, it might have enough *pieces* of memory to satisfy the request for a new array, *but* the problem is that these pieces are not continguous. Here's a solution. (This is not the last.)

Now, we can think of an array like this:

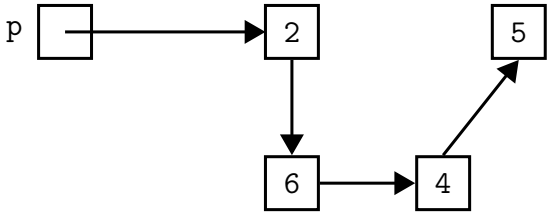


Of course in the case of a program, to *arrive* at the array, I have either an array name or in the case when the above array is in the free store, I would have a pointer to the array. Say the above is a dynamic array in the free store and I have a pointer `p` pointing to the above array:

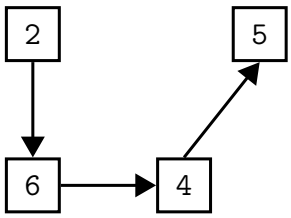


The memory used by the above array (whether static or dynamic) is contiguous (whether if it's contiguous in the current frame of execution or in the free store.) If an `int` takes 4 bytes, the above uses 16 bytes – contiguous chunk of 16 bytes. Conceptually, even on a sheet of paper, you can see that the 4 squares are contiguous.

Conceptually, suppose on a sheet of paper, I draw this:

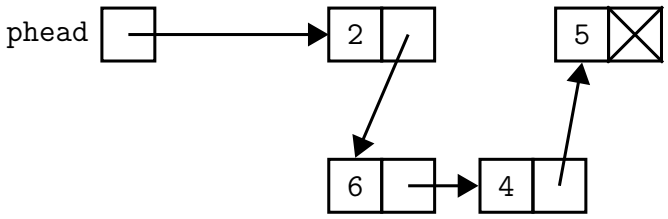


You do see that the values of the array split up into pieces and with the help of the arrows (whatever they mean) I can go from one value to another, imitating the traversal of the original array. This collection of boxes with values:



is what we call a directed graph in math and CS. I'll talk about directed graphs and undirected graphs much later. Back to linked list ...

To make the above schematic diagram even closer to C++ implementation, I'm going to draw it this way:



The array is represented (on paper) by 4 blocks (called nodes) where for each node there is an integer value and "something" in the second value to lead to the another node (of course the "something" is a pointer). The node with value 5 has a second value that does not lead to another node – it's denoted by a cross as the second value. (Of course in terms of C++, that box contains a `NULL` pointer. No surprises there.) The thing that leads to the first node (the head) of the list is now named `phead` (as in "pointer to head" if you want to think C++ ... duh).

The group of four nodes form a **singly linked list**. Each individual node is called a **singly linked node**. The `phead` is just to arrive at the first of the four nodes in the linked list. The first node of the singly linked list is called the **head** and the last node is called the **tail**. So we can call `phead` the "pointer to head". Now, although `phead` points to the head node, I will frequently say that `phead` points to the whole linked list since, after all, once you get to the head node, you can get to the rest of the nodes. This also matches our idea of a pointer that points to a dynamic array in free store: technically speaking the pointer has the address of the first value of that array, but we still think of that pointer as pointing to the whole array.

singly linked list

singly linked node

head
tail

That's on a piece of paper. Don't worry: we can implement the above idea in code.

Before we go on, let me say that in real world applications, the nodes instead

of being nodes of integers can be for instance nodes on students. The key can be for instance a student's ID – that corresponds to our integer value in a node. The satellite data such as a student's first name, last name, email address, etc. are usually not stored directly in the node. Instead the node usually contains the keys and a pointer to the satellite data. In the values of a node is copied, then only the key and the pointer-to-satellite need to be copied. I'll only focus on the key and not the satellite data since what I'm going to talk about concerns the organization of data to work quickly with keys (example: add, delete, modify). OK ... onward with data structures and algorithms on keys.

106.2.1 Singly linked node class

Here's a class for the nodes.

```
class SLNode
{
public:
    SLNode(int key, SLNode * next = NULL)
      : key_(key), next_(next)
    {}
private:
    int key_;
    SLNode * next_;
};
```

The `SL` is a shorthand for `SinglyLinked` which is too much to type.

(It's clear that ultimately, when the singly linked list of integers is done, we can convert it to template classes for modeling singly linked list of doubles, characters, strings, etc.)

It's convenient to print the nodes. When it comes to programming data structures that contains pointers, it's very useful when debugging to be able to quickly chase the pointers. So I'm going to print the nodes like this:

```
#include <iostream>

class SLNode
{
public:
    SLNode(int key, SLNode * next = NULL)
      : key_(key), next_(next)
    {}
    int key() const
    {
        return key_;
    }
    SLNode * next() const
    {
        return next_;
    }

private:
    int key_;
    SLNode * next_;
};

std::ostream & operator<<(std::ostream & cout,
                          const SLNode & node)
{
```

```
    cout << "<SLNode " << &node
         << "key:" << node.key()
         << ", next:" << node.next()
         << '>';
    return cout;
}
```

To experiment with the code we can do this:

```
// ... SLNode code from above ...

int main()
{
    SLNode n2(2, NULL);
    SLNode n6(6, NULL);

    std::cout << n2 << '\n';
    std::cout << n6 << '\n';

    return 0;
}
```

and to link the node with value 2 to the node with value 6, I can do this

```
int main()
{
    SLNode n6(6, NULL);
    SLNode n2(2, &n6);

    std::cout << n2 << '\n';
    std::cout << n6 << '\n';

    return 0;
}
```

or if you have a `set_next()` method, you can do this:

```
int main()
{
    SLNode n6(6, NULL);
    SLNode n2(2, NULL);

    n2.set_next(&n6);

    std::cout << n2 << '\n';
    std::cout << n6 << '\n';

    return 0;
}
```

(Make sure you add the `set_next()`) method.)

Of course the nodes are in the frame of `main()` and not in the free store. So go ahead and create the whole linked list from above and print out all the nodes. This is what I get this:

```
[student@localhost linkedlist] g++ tmp12345678.cpp; ./a.out
<SLNode 0x7fff3f964650 key:2, next:0x7fff3f964660>
<SLNode 0x7fff3f964660 key:6, next:0x7fff3f964670>
<SLNode 0x7fff3f964670 key:4, next:0x7fff3f964680>
<SLNode 0x7fff3f964680 key:5, next:0>
```

Exercise 106.2.1. Add enough code so that you can get the above printout. □

To imitate our earlier diagram, we create a pointer to point to the first node, the head. Let's call the pointer `phead` (again ... it means "pointer to head"). Go ahead and do it. We can now refer to the nodes of the linked list using `phead`.

Exercise 106.2.2. Go ahead and create your `phead` and set it to the right value. □

Now, instead of writing four statements to print the nodes, let's write a simple while loop. I'm going to use a pointer variable, say `p`. At the beginning, the pointer variable `p` points to the head. That means I need

```
p = phead;
```

Next I print the node that `p` points to. After that I need to get `p` to point to the next node and repeat. I stop right after printing the last node ... how do I do that? Well ... since after printing the last node, my pointer `p` move to the next and the last node has a next value of `NULL`, my `p` would have `NULL` at that point in time. Therefore if I want to stop at this point (i.e., stop the loop), it's because `p` is `NULL`.

The code would look like this:

```
SLNode * p = phead;
while (p != NULL)
{
    std::cout << (*p) << std::endl;
    p = p->next();
}
```

and the output is this:

```
[student@localhost linkedlist] g++ tmp12345678.cpp; ./a.out
<SLNode 0x7ffcf03deca0 key:2, next:0x7ffcf03decb0>
<SLNode 0x7ffcf03decb0 key:6, next:0x7ffcf03decc0>
<SLNode 0x7ffcf03decc0 key:4, next:0x7ffcf03decd0>
<SLNode 0x7ffcf03decd0 key:5, next:0>
```

We can even put that into a function:

```
void print(SLNode * p)
{
    while (p != NULL)
    {
        std::cout << (*p) << std::endl;
        p = p->next();
    }
}
```

and call this function in `main()` with

```
...
int main()
{
    ...
    print(phead);
    ...
}
```

Now, just for practice, I want to write the `print` using recursion.

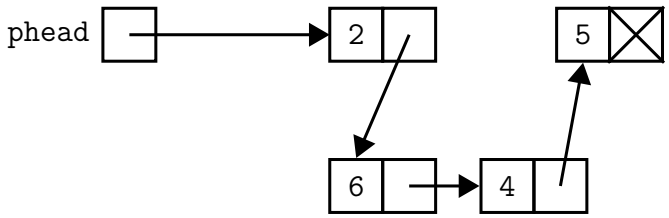
```
void print(SLNode * p)
{
    /* // while loop version
    while (p != NULL)
    {
        std::cout << (*p) << std::endl;
        p = p->next();
    }
    */

    // recursive version
    if (p != NULL)
    {
        std::cout << (*p) << std::endl;
        print(p->next());
    }
}
```

The memory used by the nodes are in the frame of `main()`. At some point we will have to put the nodes in the free store. Let's try to do that now – by hand. Later we'll have function to create nodes for us.

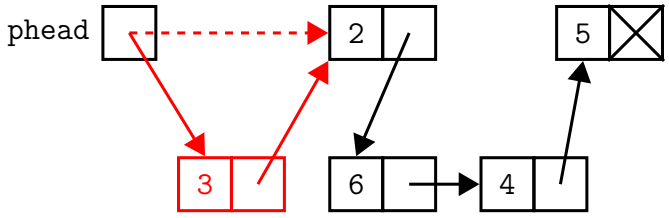
```
int main()
{
    SLNode * n5 = new SLNode(5, NULL);
    SLNode * n4 = new SLNode(4, NULL);
    SLNode * n6 = new SLNode(6, NULL);
    SLNode * n2 = new SLNode(2, NULL);

    n2->set_next(&n6);
    n6->set_next(&n4);
    n4->set_next(&n5);

    SLNode * phead = n2;

    ...
}
```

106.2.2 Insert head

We want to add values into a singly linked list. One place to add a node is at the head of the list. Here's the picture of a linked list (the same one before):



Suppose now I perform an insert head with `3` (i.e., `3` so that it becomes the new head). This means that I want this picture:



Note that there's a new node with value 3. Furthermore, the link from `phead` to the node with value 2 is broken so that `phead` points to the new node: the new node is now the head. Also, the new node (the new head) points to the node with value 2.

The pseudocode is of course

```
declare a node pointer q
allocate memory for a new node with key value of 3
    and give the address of the new node to q
set the next value of the new node to the value of phead
set the value of phead to the value of q
```

Note that the above algorithm is independent of the length of the linked list. There's no loop/repetition. You just work on one end (the head end) of the list and it doesn't matter if you have 1000000 nodes – the work is the same even if the list has 5 things. It's easy to see that the runtime is

$$O(1)$$

In terms of C++ code it would look like this:

```
SLNode * q;
q = new SLNode(3, NULL);
q->set_next(phead);
phead = q;
```

Of course the first two can be combined:

```
SLNode * q = new SLNode(3, NULL);
q->set_next(phead);
phead = q;
```

Also, the constructor allows us to set the `next` value of the node:

```
SLNode * q = new SLNode(3, phead);
phead = q;
```

This can be (finally) simplified to

```
phead = new SLNode(3, phead);
```

Remember that the right-hand side of the above is executed first so that the value of the `phead` is used on the right *before* `phead` is overwritten by the address of the new node. Now if I put the above in a function

```
void insert_head(SLNode * phead, int key)
{
    phead = new SLNode(key, phead);
}
```

it won't work ... why? Because this is passing the pointer value of `phead` in `main()` to the `phead` in `insert_head()`: this is pass by value (pointer value). I want to change the value of `phead` in `main()`. So I have to do this:

```
void insert_head(SLNode ** pphead, int i)
{
    *pphead = new SLNode(i, *pphead);
}
```

The parameter `pphead` is a pointer to `phead` in `main()`. In `main()` I have to call `insert_head()` with the *address* of `phead`. (See below).

Another way to achieve the above is to pass by reference:

```
void insert_head(SLNode *& phead, int i)
{
    phead = new SLNode(i, phead);
}
```

In this case the `phead` is a reference to the `phead` in `main()` is in `main()`. I call this function passing in `phead`.

Let's add this insert head function to our code:

```
...

void insert_head(SLNode ** phead, int i)
{
    *phead = new SLNode(i, *phead);
}

void insert_head(SLNode *& phead, int i)
{
    phead = new SLNode(i, phead);
}

int main()
{
    SLNode * phead = NULL;
    insert_head(&phead, 5);
    print(phead);

    return 0;
}
```

The output is this:

```
[student@localhost linkedlist] g++ tmp12345678.cpp; ./a.out
<SLNode 0xc01eb0 key:5, next:0>
```

Let's create the whole linked list from the previous section:

```
...
int main()
{
    SLNode * phead = NULL;
    insert_head(&phead, 5);
    insert_head(&phead, 4);
    insert_head(&phead, 6);
    insert_head(&phead, 2);
    print(phead);

    return 0;
}
```

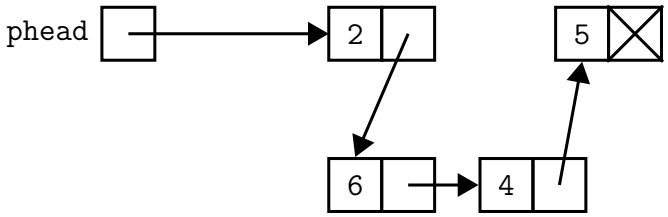
The output is this:

```
[student@localhost linkedlist] g++ tmp12345678.cpp; ./a.out
<SLNode 0x476f10 key:2, next:0x476ef0>
<SLNode 0x476ef0 key:6, next:0x476ed0>
<SLNode 0x476ed0 key:4, next:0x476eb0>
<SLNode 0x476eb0 key:5, next:0>
```

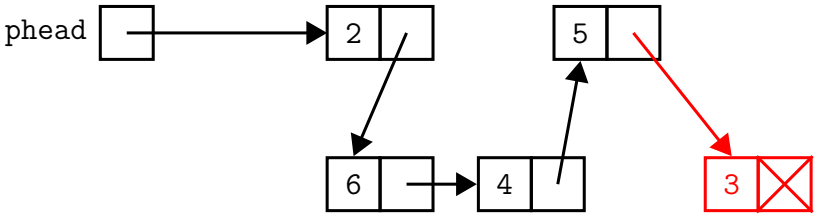
Let me say this (the obvious): the memory used by `phead` is in the frame of `main`, but the memory used by the nodes is in the free store.

106.2.3 Insert tail

Another place where integer are inserted into the list is the tail. Suppose I have this:



After I insert 3 at the tail (i.e., I make 3 the next tail), I get this picture



The pseudocode looks like this where the new value is stored in `i`:

```
declare a node pointer q
allocate memory for a new node with value i and
    store the address of the node in q
declare a node pointer r
point r to the tail: this involves looping r
    until r is pointing to the tail
set the next of the node that r is pointing to to the
    value in q
```

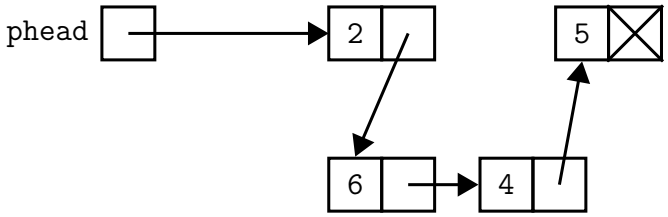
In *this* case, the runtime is clearly dependent on the length of the list since in the above computation for `r`, `r` runs through the list until the tail is reached. So this is *very* different from insert head. This means that if there are n nodes, then `r` must run through all the addresses of the n nodes. The runtime is clearly

$$O(n)$$

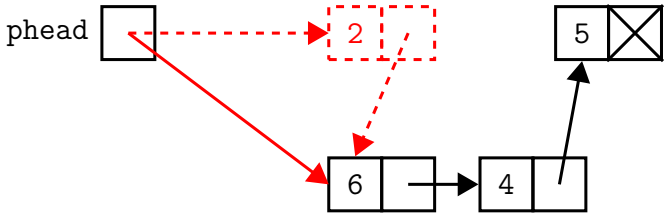
I'll leave the implementation to you. Simplify the code. Obviously ... test your code.

106.2.4 Delete head

Here's our linked list again:



By a delete (or remove) head, I mean that I will get this after the operation:



Two things must happen. You see that the head node to be removed from the linked list is removed from the list: the `phead` points to the node with value 6. But another really important thing is that the node with value 2 must be given back to the free store. (Don't forget the nodes are now in the free store, not in the frame of `main`.)

```
let q be a pointer that points to the head, i.e., give the
    value in phead to q
let phead point to the node that q points to, i.e., give the
    next value of the node q points to to phead
deallocate the node that q points to
```

Be careful ... have you considered the case where there's no head? i.e., what should happen when you delete head when the linked list is empty? If you want the delete to fail quietly, then you do nothing:

```
if phead is not NULL:
    let q be a pointer that points to the head, i.e., give
        the value in phead to q
    let phead point to the node that q points to, i.e., give
        the next value of the node q points to to phead
    deallocate the node that q points to
```

However you might want to throw an exception (or return a boolean value of true to indicate a successful delete and false if there's a failure.) In the case you want to do this:

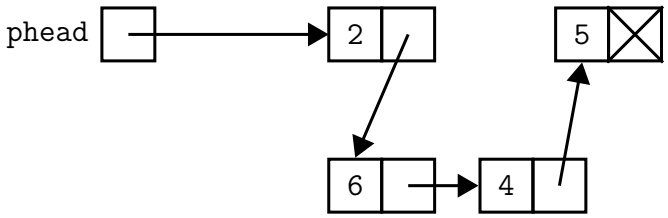
```
if phead is not NULL:
    let q be a pointer that points to the head, i.e., give
        the value in phead to q
    let phead point to the node that q points to, i.e., give
        the next value of the node q points to to phead
    deallocate the node that q points to
else:
    throw an exception (some kind of underflow exception)
```

Exercise 106.2.3. What is the runtime? □

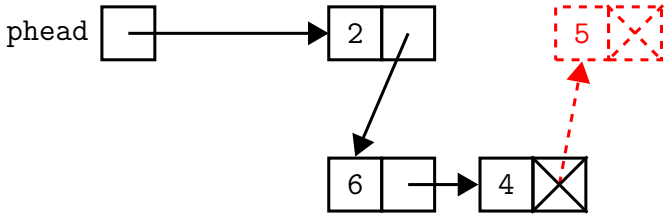
Exercise 106.2.4. Write a function `delete_head()` so that calling `delete_head(&phead)` from `main()` (like before) removes the head node. (Test it.) Write another version so that you call `delete_head(phead)` instead. □

106.2.5 Delete tail

It should be clear what you should do here. If I start with this:



and I do delete tail, then I get this:



The meaning of the diagram should be clear.

Exercise 106.2.5. Write down the pseudocode/algorithm. What is the runtime? Implement and test it. □

Exercise 106.2.6. Suppose `p` points to a node in the singly linked list. Write a function so that `delete(&phead, p)` deletes the above node from the linked list. □

106.2.6 Find

Exercise 106.2.7. Write a function `find()` such that `find(phead, key)` returns the address of the first node (going from head to tail) that has a key with value of `key`. What is the runtime? □

Exercise 106.2.8. Write a function `find()` such that `find(p, q, key)` returns the address of the first node that has key value of `key` going from the node at address `p` up to but not including the node with address `q`. □

106.2.7 Exercises

Exercise 106.2.9. Write a function `clear()` such that `clear(phead)` removes all the values in the linked list.

Exercise 106.2.10. Write a function `copy()` such that on calling `copy(phead)`, the linked list that `phead` points to is cloned and the pointer to head of the clone is returned. In other words after

```
clear(phead2);
phead2 = copy(phead1)
```

`phead1` and `phead2` will point to linked lists with the same values. What is the runtime? □

Exercise 106.2.11. Write a function `size()` such that `size(phead)` returns the number of values in the linked list. What is the runtime? □

Exercise 106.2.12. Write a function `is_empty()` such that `is_empty(phead)` returns the true exactly when there is no value in the linked list. What is the runtime? □

Exercise 106.2.13. Write a function `concat()` such that on calling `concat(&phead1, phead2)` the values in the linked list that `phead2` points to to used to extend the linked list that `phead1` points to. For practice, write another version that allows you to do `concat(phead1, phead2)`. Write an `operator+=()` that does the same thing. □

Exercise 106.2.14. Write an `operator+()` so that on calling `operator+(phead1, phead2)` a new list is built with values from the above two lists and the head address is returned. □

Exercise 106.2.15. Write a function `prepend()` such that on calling `prepend(&phead1, phead2)` the values in the linked list that `phead2` points to to used to extend (at the head) the linked list that `phead1` points to. For instance if `phead1` points to 1,2,3 and `phead2` points to 4,5,6, then after calling the function, `phead1` points to 4,5,6,1,2,3. □

Exercise 106.2.16. Write a function `count()` so that `count(phead, key)`

returns the number of times `key` appears in the linked list. □

Exercise 106.2.17. Linked list are not strong in random access, i.e., returning `phead[k]`, the k–th value in the list linked is slow. Why? What is the runtime? Implement it anyway. Linked list are good at sequential access. In the case of singly linked list, they are good when the sequential access in one direction. □

Exercise 106.2.18. Write a function `reverse()` that reverses the values in the linked list. For instance is the singly linked list has values 5,7,1,3 (head to tail), then after calling your reverse function, the singly linked list has value 3,1,7,5 (head to tail). The runtime must as as fast as possible (it should be $O(n)$ where n is the number of values in your linked list) and you must use the least number of variables. (Go to solution.) □

Exercise 106.2.19. Write a function that essentially performs one pass of the selection sort on a singly linked list. In other words, after calling our function the smallest value that appears earliest in the singly linked list is at the head. For instance if the singly linked list is 5,7,1,3 (head to tail), then after your function is executed, the singly linked list is 1,7,5,3 (head to tail). There are two ways to do this:

1. Swap values in the singly linked list and don't change any pointers in the singly linked list nodes.
2. Change some pointers in the singly linked list nodes and do not swap any values in the singly linked list.

Do both. What is the runtime for both? (The second version is obviously more complicated and if this question appears in a job interview, it's the second one they are asking for.) □

Solutions

Solution to Exercise 106.2.18.

To reverse a singly linked list

```
    o-->o-->o-->o-->o-->o
```

You want to iteratively achieve this:

```
    o<--o<--o   o-->o-->o    p = phead of DONE
            p   q            q = phead of TODO
```

until you get

```
    o<--o<--o<--o<--o<--o
                        p   q
```

Initially you probably want this

```
    o-->o-->o-->o-->o-->o
 p  q
```

To go from

```
    o<--o<--o   o-->o-->o
            p   q
```

to

```
    o<--o<--o<--o   o-->o
                p   q
```

you probably want

```
    o<--o<--o   o-->o-->o
            p   r   q
```

i.e.

```
    r = q
    q = r->next_
    r->next_ = p
    p = r
```

So the code is

```
reverse(phead):
    p = NULL
    q = phead

    while q is not NULL:
        r = q
        q = r->next_
        r->next_ = p
        p = r
```

Note that `phead` must be set to the last value of `p`:

```
reverse(phead):
    p = NULL
    q = phead

    while q is not NULL:
        r = q
        q = r->next_
        r->next_ = p
        p = r

    phead = p // phead is changed, therefore pass by reference
```

Or you can return `p`:

```
reverse(phead):
    p = NULL
    q = phead

    while q is not NULL:
        r = q
        q = r->next_
        r->next_ = p
        p = r

    return p
```

and use `reverse` like this:

```
phead = reverse(phead)
```

Working example:

```
#include <iostream>

class Node
{
public:
    Node(int key, Node * next=NULL)
        : key_(key), next_(next)
    {}
    int key_;
    Node * next_;
};

void print(Node *p)
{
    while (p != NULL)
    {
        std::cout << p->key_ << ' ';
        p = p->next_;
    }
    std::cout << '\n';
}

void clear(Node * p)
```

```
{
    while (p != NULL)
    {
        Node * q = p;
        p = q->next_;
        delete q;
    }
}

Node * reverse(Node * phead)
{
    Node * p = NULL;
    while (phead != NULL)
    {
        Node * r = phead;
        phead = r->next_;
        r->next_ = p;
        p = r;
    }
    return p;
}

int main()
{
    // 2 3 5 7 11
    Node * p = new Node(11);
    p = new Node(7, p);
    p = new Node(5, p);
    p = new Node(3, p);
    p = new Node(2, p);
    print(p);

    p = reverse(p);
    print(p);

    clear(p);
    return 0;
}
```

□

106.2.8 Singly linked list class

I already have a class for the the nodes. Most of the functions from previous section is enough for list processing.

But now I also want a class to represent the whole singly linked list. What should go into the class?

Well first of all, since a linked list represents the whole chain of all singly linked nodes, the whole linked list. I definitely want to access the head since from the head of the linked list, I can get all the nodes. Therefore ... clearly my singly linked list objects must all have a pointer to the head of the linked list. So here's the start of our singly linked list class:

```
// SLNode class here

class SLList
{
public:
    SLList()
    : phead_(NULL)
    {}

private:
    SLNode * phead_;
};
```

Taking the cue from `std::vector`, I want to quickly find out how big my singly linked list it. So I'll keep a size measure in the class as well. Of course if you never ever ask for the size of the linked list, then `size_` is redundant:

```
// SLNode class here

class SLList
{
public:
    SLList()
    : phead_(NULL), size_(0)
    {}

private:
    SLNode * phead_;
    int size_;
};
```

Of course you increment and decrement `size_` depending on whether you're adding or removing nodes to the linked list.

Now if you think about it, you'll see that in fact when you're completely done with the above class, for your cleanup, the `SLNode` class should probably be nested inside `SLList` class. Why? Because you probably want to do something like

```
SLList list;
list.insert_head(5);
std::cout << list << '\n';
```

i.e., you would say "add this or that integer to my list", "remove this or that integer from my list", "print the list", etc. In other words, you almost never refer to the `SLNode`. In fact, if you're really sure you (or someone else) don't use `SLNode` outside the above file, then you might even want to have the `SLNode` in a *private* section of `SLList`:

```
class SLList
{
private:
    class SLNode
    {
        ...
    };
...
};
```

I hope you realize that all the functions you wrote earlier won't work anymore since the name `SLNode` is hidden. How would you fix that with the least amount of code?

Exercise 106.2.20. Answer the above question. Test your idea. □

Of course the first thing to do is to print your linked list:

```
class SLList
{
    ...
};

std::ostream & operator<<(std::ostream & cout,
                          const SLList & list)
```

```
{
    // FILL IN THE BLANKS
    return cout;
}
```

The next few things to do is pretty standard ...

Exercise 106.2.21. Since the `SLList` class contains pointers, clearly you have to write your own

- destructors
- copy constructor, and
- `operator=()`.

Do it now. □

Exercise 106.2.22. Add code to the class so that you can do

```
SLList list;
list.insert_head(5);
list.push_front(5);
```

□

Exercise 106.2.23. Add code to the class so that you can do

```
SLList list;
list.insert_head(5);
std::cout << list.get_head() << '\n'; // 5 is printed
std::cout << list.front() << '\n';    // 5 is printed
list.front() = 42;    // change head value to 42
```

□

Exercise 106.2.24. Add code to the class so that you can do

```
SLList list;
list.insert_tail(4);
std::cout << list.get_tail() << '\n'; // 4 is printed
std::cout << list.back() << '\n';      // 4 is printed
list.back() = 42;   // change tail value to 42
```

□

Exercise 106.2.25. Add code to the class so that you can do

```
SLList list;
list.insert_head(5);
list.delete_head();
```

□

Exercise 106.2.26. Add code to the class so that you can do

```
SLList list;
list.insert_tail(5);
list.push_back(5);
list.delete_tail();
list.pop_back()
```

□

Exercise 106.2.27. Add code so that if `list` is an `SLList` object, you can do `list.size()`. □

Exercise 106.2.28. Write code so that you can do `list.clear()` so that after doing that the linked list is empty. □

Exercise 106.2.29. Once you're done with the above, change it to a class template. □

Floyd's tortoise and hare algorithm

Exercise 106.2.30. Floyd's tortoise and hare algorithm. Do the following experiment: Create a singly linked list. Let `fast` and and `slow` point to the the head of your linked list. Move `slow` forward by one node and `false` forward by one node. Print the nodes that `slow` and `fast` are pointing to.

cycle

Exercise 106.2.31. You have a singly linked list that is corrupted: one of the nodes is pointing backward to a previous node. This creates a **cycle** in your singly linked list. How would you detect the presence of (or lack of) cycles in a linked list?

Exercise 106.2.32. Compute the address of the singly linked node in the "middle" of a singly linked list. If there is an odd number of nodes, it's clear what I mean by the "middle node". If there's an even number of nodes, there are two nodes "in the middle". In this case, compute the address of the middle node on the right. For instance if your singly linked list has values $1, 2, 3, 4, 5{,}6$ (from head to tail), you

should compute the address of 4.

106.2.9 Iterators

Add iterators to your `SLList` class:

```
class SLList
{
public:
    class iterator
    {
    private:
        SLNode * p;
    };
private:
}
```

Of course an `SLList::iterator` object is more or less a pointer to a value in the singly linked list.

Exercise 106.2.33. Add a `begin()` method to `SLList` so that it returns an iterator object that points to the head node. When references the iterator object returns a reference to the key value of the node that the iterator points to.

```
SLList list;
list.insert_head(5);
list.insert_tail(100);
typename SLList::iterator p = list.begin();
std::cout << (*p) << std::endl; // prints 5
*p = 42;                        // change head key value to 42
*(list.begin()) = 43;           // change head key value to 43
```

Exercise 106.2.34. Add just enough code to your iterator class so that you can do this:

```
SLList list;
list.insert_head(5);
list.insert_tail(100);
typename SLList::iterator p = list.begin();
++p;                            // p points to tail
std::cout << (*p) << std::endl; // prints 100
*p = 42;                        // change tail key value to 42
```

Exercise 106.2.35. Now add code so that you can also do this `p++`. What is the difference between and pre- and post-increment?

Exercise 106.2.36. Add just enough code to your iterator class so that you can do this:

```
SLList list;

// list: 5 -> 100 -> 200 -> 300
list.insert_head(5);
list.insert_tail(100);
list.insert_tail(200);
list.insert_tail(300);

// Prints 5 100 200 300
for (typename SLList::iterator p = list.begin();
     p != list.end(); ++p)
{
    std::cout << (*p) << ' ';
}
std::cout << std::endl;
```

Now add a constant iterators to your `SLList` class:

```
class SLList
{
public:
    class iterator
    {
    private:
        SLNode * p;
    };

    class const_iterator
    {
    private:
        SLNode * p;
    };

private:
}
```

Exercise 106.2.37. Write enough code so that you can do this:

```
void print(const SLList & list)
{
    for (typename SLList::const_iterator p = list.begin();
         p != list.end(); ++p)
```

```
    {
        std::cout << (*p) << ' ';
    }
    std::cout << std::endl;
}

int main()
{
    SLList list;

    // list: 5 -> 100 -> 200 -> 300
    list.insert_head(5);
    list.insert_tail(100);
    list.insert_tail(200);
    list.insert_tail(300);
    print(list);           // prints 5 100 200 300

    ...
}
```

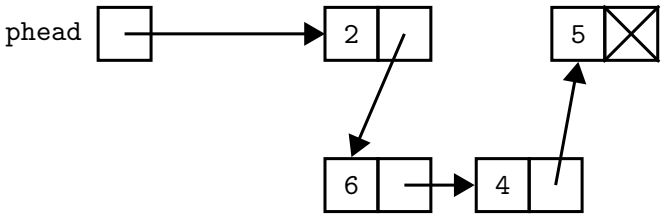
A constant iterator cannot change the value that iterator points to (or rather it cannot change the value that *the pointer in the iterator* points to). This means that if `p` is an `SLList` constant iterator, `*p` should not return a reference, but rather a constant reference. □

106.3 Sentinel node

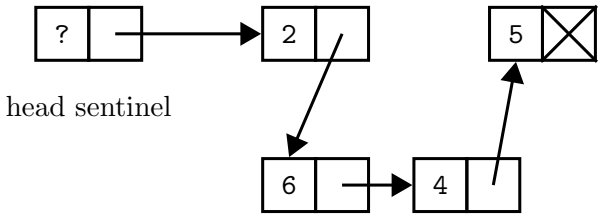
debug: sentinel-node.tex

sentinel node

A **sentinel node** is a dummy node. It does not hold data. Let me show you what I mean. Look at this singly linked list:



Suppose I add a dummy node like this:



Note that that key value of the sentinel node is not used. I'm using the next pointer of the sentinel node as the pointer to the head node (if there's any).

head sentinel

This dummy node is sometimes called the **head sentinel**. You can think of the head sentinel node as a fake node. The main purpose of this node is to point to the head node of the linked list (if the list is non-empty).

Instead of using a head sentinel node in a singly linked list object:

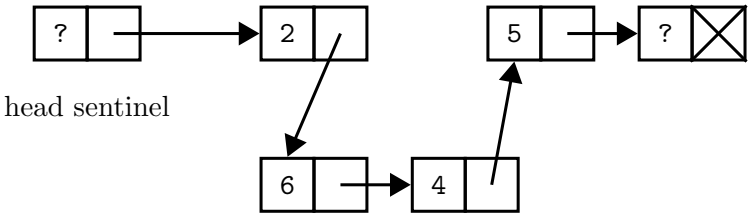
```
class SLList
{
public:
    ...
private:
    SLNode headsentinel_;
};
```

another implementation is to use a pointer to a head sentinel node where the head sentinel node is in the heap:

```
class SLList
{
public:
    ...
private:
    SLNode * pheadsentinel_;
};
```

tail sentinel

You can also attach a **tail sentinel** to the end of the singly linked list. Here's an example where I have both head and tail sentinel nodes:



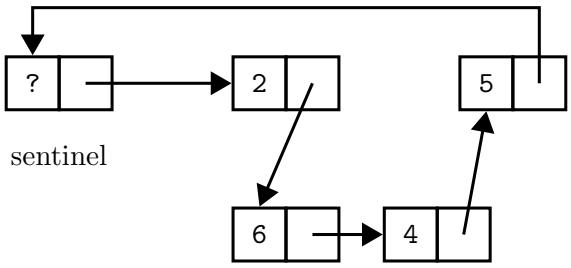
The tail sentinel can then be used as an end-of-singly-linked-list node. So for a computation that involves iterating over a singly-linked list, the head sentinel node helps you begin the iteration while the tail sentinel node helps you stop the iteration.

Here's an implementation of a singly linked list class where both head and tail sentinel nodes are stored in the heap:

```
class SLList
{
public:
    ...
private:
    SLNode * pheadsentinel_;
    SLNode * ptailsentinel_;
};
```

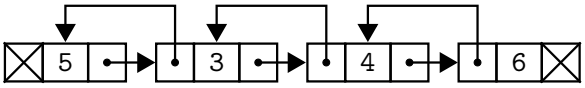
Sentinel nodes can simplify your code by providing greater uniformity for linked list computations. For instance if `p` is an `SLNode` pointer that runs through all the nodes in your linked list, when using sentinel nodes, `p` is always pointing to a node and is not `NULL`. When you do not use sentinel nodes, if your linked list is empty, `p` will begin with `NULL`. If your linked list is not empty, the last value of `p` might be `NULL`. I'll be using sentinel nodes in the next section.

It's also possible to implement a linked list where there is one sentinel that acts as both the head and tail sentinel:


?
2
5
sentinel
6
4

106.4 Doubly linked list debug: doublylinkedlist.tex

A doubly linked list is similar to a singly linked list: For a singly linked list, you start at head and at every node you can go to the next node (except when the pointer is NULL). For a doubly linked list, you can start at the head or tail and at every node you can go forward and backward. Here's an example:




```
class DLNode
{
public:
    DLNode(DLNode * prev = NULL, DLNode * next = NULL)
    : prev_(prev), next_(next)
    {}
    DLNode(int key, DLNode * prev = NULL, DLNode * next = NULL)
    : key_(key), prev_(prev), next_(next)
    {}
private:
    int key_;
    DLNode * prev_;
    DLNode * next_;
};
```

The double linked list class would contain two pointers, one pointing to the head node and another pointing to the tail node. If the list is empty, these two pointers are set to NULL.

In the case when the doubly linked list uses sentinel nodes, you would have

```
class DLList
{
public:
    DLList()
    : pheadsentinel_(new DLNode), ptailsentinel_(new DLNode)
    {
        pheadsentinel_->next_ = ptailsentinel_;
        ptailsentinel_->prev_ = pheadsentinel_;
    }
private:
    DLNode * pheadsentinel_;
    DLNode * ptailsentinel_;
};
```

106.4.1 Insertion

Here's insert head:

```
void DLList::insert_head(int v)
{
    DLNode * p = new DLNode(v, pheadsentinel_, pheadsentinel_->next_);
    p->prev_->next_ = p;
    p->next_->prev_ = p;
}
```

Note the symmetry in the last two statements – think of `p` as the center of attraction so the node before and after `*p` are linked up with `*p`. Also, note that it's because of the sentinel nodes that you get such a clean implementation. For instance if your doubly-linked class has `phead` (instead of `pheadsentinel`), you would probably have the following:

```
void DLList::insert_head(int v)
{
    DLNode * p = new DLNode(v, NULL, phead_);
    if (phead_ != NULL)
    {
        phead_->prev_ = p;
    }
    phead_ = p;
}
```

For our implementation of the doubly-linked list class, I will be using sentinel nodes. I'll leave it to you to write the version without sentinel nodes so that you see the difference.

There are other ways to implement the above insert head. Here's another way that is not as simple and symmetric as the above (and is a lot uglier, in my opinion)

```
void DLList::insert_head(int v)
{
    DLNode * p = new DLNode(v, pheadsentinel_, pheadsentinel_->next_);
    pheadsentinel_->next_ = p;
    p->next_->prev_ = p;
}
```

There's also a dependency on `pheadsentinel->next_` in the second statement.

So what?

Well you might write this:

```
void DLList::insert_head(int v)
{
    DLNode * p = new DLNode(v, pheadsentinel_, pheadsentinel_->next_);
    pheadsentinel->next_->prev_ = p;
    pheadsentinel->next_ = p;
}
```

or this:

```
void DLList::insert_head(int v)
{
    DLNode * p = new DLNode(v, pheadsentinel_, pheadsentinel_->next_);
    pheadsentinel->next_ = p;
    pheadsentinel->next_->prev_ = p; // oops
}
```

The second of these two is incorrect. (See it?)

However the second and third statements of the first insert head can be swapped:

```
void DLList::insert_head(int v)
{
    DLNode * p = new DLNode(v, pheadsentinel_, pheadsentinel_->next_);
    p->prev_->next_ = p;
    p->next_->prev_ = p;
}
```

The insert head

```
void DLList::insert_head(int v)
{
    DLNode * p = new DLNode(v, pheadsentinel_, pheadsentinel_->next_);
    p->prev_->next_ = p;
    p->next_->prev_ = p;
}
```

can be easily generalized to

```
void DLList::insert_after(DLNode * q, int v)
{
    DLNode * p = new DLNode(v, q, q->next_);
    p->prev_->next_ = p;
    p->next_->prev_ = p;
}
```

So in fact `insert_head` can use `insert_after`:

```
void DLList::insert_head(int v)
{
    insert_after(pheadsentinel_, v);
}
```

Note that you obviously don't want to insert after the tail sentinel! So let's prevent that from happening:

```
void DLList::insert_after(DLNode * q, int v)
{
    if (q == psentineltail_)
    {
        throw InsertAfterTailError();
    }
    DLNode * p = new DLNode(v, q, q->next_);
    p->prev_->next_ = p;
    p->next_->prev_ = p;
}
```

where `InsertAfterTailError` is some exception class. Note that the boolean check is redundant if `insert_after` is never called with `q` = `psentineltail_`. Linked list classes are meant to work extremely fast. So like arrays where out-of-bound checks are not executed, it's also reasonable for a linked list class not to check if an insert is performed after the tail sentinel. If so, this should be documented in the linked list class.

The mirror image of insert head is

```
void DLList::insert_tail(int v)
{
    DLNode * p = new DLNode(v, ptailsentinel_->prev_, pheadsentinel_);
    p->prev_->next_ = p;
    p->next_->prev_ = p;
}
```

See the beauty and the symmetry?

Note that this can be generalized to

```
void DLList::insert_before(DLNode * q, int v)
{
    if (q == pheadsentinel_)
    {
        throw InsertBeforeHeadError();
    }
    DLNode * p = new DLNode(v, q->prev_, q);
```

```
    p->prev_->next_ = p;
    p->next_->prev_ = p;
}
```

and so

```
void DLList::insert_tail(int v)
{
    insert_before(psentineltail_, v);
}
```

But in fact ... note that

```
void DLList::insert_before(DLNode * q, int v)
{
    if (q == pheadsentinel_)
    {
        throw InsertBeforeHeadError();
    }
    insert_after(q->prev_, v);
}
```

Nice and beautiful right?

106.4.2 Deletion

Now for node deletion:

```
void DLList::delete_at(DLNode * p)
{
    if (p == pheadsentinel_ || p == ptailsentinel_)
    {
        throw DeleteSentinelError();
    }
    p->prev_->next_ = p->next_;
    p->next_->prev_ = p->prev_;
    delete p;
}
```

This can then be used for

```
void DLList::delete_head(DLNode * p)
{
    if (is_empty())
    {
        throw UnderflowError();
    }
    else
    {
        delete_at(pheadsentinel_->next_);
    }
}
```

and

```
void DLList::delete_tail(DLNode * p)
{
    if (is_empty())
    {
        throw UnderflowError();
    }
    else
    {
        delete_at(ptailsentinel_->prev_);
    }
}
```

106.4.3 Destructor, copy constructor, assignment operator

Of course with the above methods to acquire resource (memory from the heap), we need to rewrite

1. the destructor
2. the copy constructor, and
3. the assignment operator

Here's the destructor:

```
DLList::~DLList()
{
    /* The following idea works for singly-linked list:
    DLNode * p = pheadsentinel_;
    while (p != NULL)
    {
        DLNode * q = p->next_;
        delete p;
        p = q;
    }
    */

    DLNode * p = pheadsentinel_;
    while (p != ptailsentinel_)
    {
        p = p->next_;
        delete p->prev_;
    }
    delete ptailsentinel_;
}
```

Can we generalize this? Yes, for instance we can have a method that removes a "section" of nodes, i.e., a contiguous block of nodes. I'll call it `DLList::delete(begin, end)` where `begin` and `end` are addresses and the nodes deleted are from address `begin` up to *but not including* address `end`. This can also be used for instance in a method that *clears* the doubly-linked list so that it becomes empty. For use by the destructor, since the head and tail sentinels need not be joined up, I'll have a `delete_` that only deletes nodes and not join up the two nodes on both ends:

```
void DLList::delete_(DLNode * begin, DLNode * end)
{
    // If necessary, throw an exception if p is pheadsentinel_.
    while (begin != end)
    {
        begin = begin->next_;
        delete begin->prev_;
    }
```

```
}
```

Then

```
DLList::~DLList()
{
    delete_(pheadsentinel_, ptailsentinel_);
    delete ptailsentinel_;
}
```

```
void DLList::delete(DLNode * begin, DLNode * end)
{
    if (begin != end)
    {
        DLNode * begin_prev = begin->prev_;
        delete_(begin, end);
        begin_prev->next_ = end;
        end->prev_ = begin_prev;
    }
}
```

```
void DLList::clear()
{
    delete(pheadsentinel_->next_, ptailsentinel_);
}
```

Here's the copy constructor:

```
DLList::DLList(const DLList & list)
    : pheadsentinel_(new DLNode), ptailsentinel_(new DLNode)
{
    pheadsentinel_->next_ = ptailsentinel_;
    ptailsentinel_->prev_ = pheadsentinel_;

    DLNode * p = list.pheadsentinel_->next_;
    while (p != list.ptailsentinel_)
    {
        insert_tail(p->key_);
        p = p->next_;
    }
}
```

Note that this uses `insert_tail` so that the new node created is continually connected to the tail sentinel, which is redundant except for the tail node. Here's another version:

```
DLList::DLList(const DLList & list)
```

```
    : pheadsentinel_(new DLNode), ptailsentinel_(new DLNode)
{
    DLNode * p = list.pheadsentinel_->next_;
    DLNode * q = pheadsentinel_;
    while (p != list.ptailsentinel_)
    {
        q->next_ = new DLNode(p->key_, q, NULL);
        p = p->next_;
        q = q->next_;
    }
    q->next_ = ptailsentinel_;
    ptailsentinel_->prev_ = q;
}
```

Of course the copy constructor is very similar to `operator=`.

```
const DLList & DLList::operator=(const DLList & list)
{
    if (this != &list)
    {
        // 1. Copy key values from list to *this
        // 2. If list has more values, extend *this with values from list
        // 3. If *this has more values, delete extra nodes from *this.
        DLNode * p = list.pheadsentinel_->next_;
        DLNode * q = pheadsentinel_;
        while (p != list.ptailsentinel_ && q != ptailsentinel_)
        {
            q->key_ = p->key_;
            p = p->next_;
            q = q->next_;
        }

        if (p == list.ptailsentinel_)
        {
            delete(q, ptailsentinel_);
        }
        else if (q == ptailsentinel_)
        {
            while (p != list.ptailsentinel_)
            {
                insert_tail(p->key_);
                p = p->next_;
            }
        }
    }
    return (*this);
}
```

Note that the last part of the above method that inserts the values of a section of

a list to a point of another list is helpful. Here's `insert_after` inserts a sequence of values and not just a single value:

```
DLList::insert_after(DLNode * p, DLNode * begin, DLNode * end)
{
    DLNode * t = p;
    while (begin != end)
    {
        t->next_ = new DLNode(begin->key_, t, NULL);
        t = t->next_;
        begin = begin->next_;
    }
    t->next_ = p->next_;
    p->next_->prev_ = t;
}
```

With this, the copy constructor becomes

```
DLList::DLList(const DLList & list)
    : pheadsentinel_(new DLNode), ptailsentinel_(new DLNode)
{
    insert_after(pheadsentinel_, list.pheadsentinel_->next_, list.ptailsentinel_);
}
```

and the `operator=` becomes

```
const DLList & DLList::operator=(const DLList & list)
{
    if (this != &list)
    {
        // 1. Copy key values from list to *this
        // 2. If list has more values, extend *this.
        // 3. If *this has more values, delete part of *this.
        DLNode * p = list.pheadsentinel_->next_;
        DLNode * q = pheadsentinel_;
        while (p != list.ptailsentinel_ && q != ptailsentinel_)
        {
            q->key_ = p->key_;
            p = p->next_;
            q = q->next_;
        }

        if (p == list.ptailsentinel_)
        {
            delete(q, ptailsentinel_);
        }
        else if (q == ptailsentinel_)
        {
            insert_after(ptailsentinel_->prev, p, list.ptailsentinel_);
```

```
        }
    }
    return (*this);
}
```

Now for search, which is basically linear search. Note doubly linked lists are *not* really meant for search since the runtime is slow.

```
DLNode * DLList::find(int v)
{
    DLNode * p = pheadsentinel_;
    DLNode * q = list.pheadsentinel_->next_;
    while (q != list.ptailsentinel_)
    {
        p->next_ = new DLNode(q->key_, p, NULL);
        q = q->next_;
        p = p->next_;
    }
    p->next_ = ptailsentinel_;
    ptailsentinel_.prev_ = p;
}
```

You can also find in the "reverse direction":

```
DLNode * DLList::find(int v)
{
    DLNode * p = ptailsentinel_->next_;
    while (p != pheadsentinel_)
    {
        if (p->key_ == v)
        {
            return p;
        }
        p = p->prev_;
    }
    return NULL;
}
```

Exercise 106.4.1. What will happen if `pheadsentinel_->prev_` is set to `pheadsentinel_`? In other words, the previous pointer of the head sentinel node is pointing to itself. And what will happen if the next pointer of the tail sentinel node is pointing to itself?

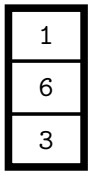
Exercise 106.4.2. Implement the insertion sort on a doubly linked list.

106.5 Circular linked list debug: circularlinkedlist.tex

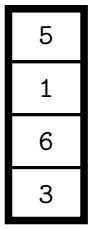
[TODO: circular buffer, Josephus problem]

106.6 Stack debug: stack.tex

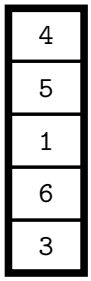
Recall (see CISS240/245) that a stack is an ADT that looks like the container for plates at a buffet. Here's a stack with three values:



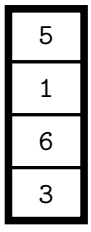
If I put a 5 into the stack, it looks like this:



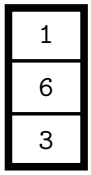
If I put a 4 into the stack, then it looks like this:



If I take a value out of the stack I would get 4 and the stack would look like this:



Note that the value at the top of the stack is the one removed. Now if I remove a value again, the stack would look like this:



A stack is self-organizing in the sense that you can put values into the stack, but you do not tell the stack *where* to put the value. The stack will decide where to put it. Also, you can retrieve a value from the stack, but you don't say which value. The ordering of the values is such that the value taken out from the stack is the last value put into the stack. This is just like a stack of plates: the plate to take off the stack is the one on top and is the last plate placed into the stack.

Putting a value into the stack is called **pushing** a value onto the stack. The value that a user can see when he's looking at a stack is the value on top of the stack. This is a technical term: the value on top of the stack is called the **top** of the stack. Removing a value out of the stack is called **popping** the stack. Frequently, the value popped off the stack is returned.

What else can you do with a stack?

- Asking for the size of the stack
- Asking if the stack is empty
- Clearing the stack

Etc. If you pop an empty stack, you should probably thrown an exception.

You can (and should) think of a stack as a memory device. When you push something onto the stack, you're saying to the stack "Hang on to this please. I'll want it back later." When you take something out of a stack, you're basically saying to the stack "Give me the last thing I handed to you."

Now for the implementation of stack ...

Note that the stack is very "linear". So it's not too surprising that you can implement it using arrays (dynamic or static) and linked list. Now note that the operation on a stack is at "one end" of the stack.

Exercise 106.6.1. If you implement a stack using an array, which end of the array should be used (the slot at index 0 of the opposite) for the top of the stack? Why? What is the runtime for push and pop? (If the maximum size of the stack is small, then an array will do.) □

Exercise 106.6.2. If the stack is huge, then a linked list can be used. Should we use a singly or a doubly linked list? Why? What is the runtime for push and pop? □

Exercise 106.6.3. Implement the stack (to contain integers) so that you can do this:

```
Stack stack;
std::cout << stack.size() << '\n';     // prints 0
stack.push(3);
std::cout << stack << '\n';            // prints [3]
stack.push(6);
std::cout << stack << '\n';            // prints [6, 3]
stack.push(1);
std::cout << stack << '\n';            // prints [1, 6, 3]
std::cout << stack.top() << '\n';      // prints [1, 6, 3]
std::cout << stack.is_empty() << '\n'; // prints 0
int t = stack.pop();
std::cout << t << '\n';                // prints 1
std::cout << stack << '\n';            // prints [6, 3]
```

Exercise 106.6.4. Once you're done with the above, change it to a class template so that the above code becomes:

```
Stack< int > stack;
...
```

You can also have a stack of doubles:

```
Stack< double > stack;
...
```

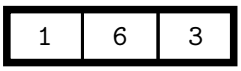
Exercise 106.6.5. Of course there are standard class operations you want:

```
Stack< int > stack0;
Stack< int > stack1(stack0);
std::cout << (stack0 == stack1) << std::endl;
stack1 = stack0;
```

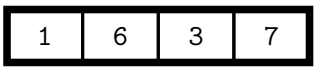
106.7 Queue debug: queue.tex

A queue (see CISS240/245) is an ADT where you can put things into the container and you can take things out of it. The operation of putting a value into a queue is called enqueue and the operation of taking a value out of a queue is called dequeue. Like the stack, it's self-organizing: you don't have to tell the queue where to put a value and you don't have to tell the queue which one to remove.

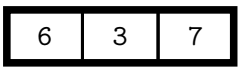
Suppose this is a queue:



A queue has a front and a back. Let's say for the above diagram the value 1 is the front and the 3 is the back. If I enqueue with 7, the queue becomes:



If I dequeue, the 1 is removed (frequently returned) and the queue becomes



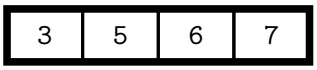
Note that the value removed is the value that was in the queue the longest. So we say that a queue is a **first-in-first-out**, **FIFO**, data structure.

first-in-first-out
FIFO

You see queues everywhere: a network router processes messages that are stored in a queue. A webserver processes requests that are stored in a queue. The OS running your laptop will run processes that are stored in a queue. I/O requests are also stored in queues. Etc.

Note that since you have to operate a queue on both ends, you should use a doubly linked list and not a singly linked list. (An array implementation is possible if the maximum length of the queue is small and fixed.)

Frequently, jobs entering a queue has a priority number that allows the job to jump ahead. For instance the following is a queue of jobs and the numbers are the priority numbers where low means high priority:



So if I enqueue 5, the 5 will enter the queue at the end but will jump ahead of the job with priority 7, jump ovrer the jobn with priority 6, but will not jump ahead of the job with priorityt 5 that was already there:

Such things are called priority queues. We'll come back to this later when we look at heaps.

Exercise 106.7.1.

- If you use an array to implement a queue, what is the runtime of enqueue and dequeue?
- If you use a singly linked list to implement a queue, what is the runtime of enqueue nad dequeue? (Remember that in this case you have to say where is the front and where is the end of the queue with respect to the list.)
- If you use a doubly linked list to implement queue, what is the runtime of enqueue nad dequeue?

□

Exercise 106.7.2.

- If you use an array to implement a priority queue, what is the runtime of enqueue and dequeue?
- If you use a singly linked list to implement a priority queue, what is the runtime of enqueue nad dequeue? (Remember that in this case you have to say where is the front and where is the end of the queue with respect to the list.)
- If you use a doubly linked list to implement a priority queue, what is the runtime of enqueue nad dequeue?

□

106.8 Double–ended queue debug: deque.tex

A double-ended queue (deque) is a data structure that contains features of a stack and a queue: it's just a doubly-linked list. You can insert and delete values on both ends. That's why it's called a *double* ended queue. That's all.

106.9 C++ STL singly linked list: `std::forward_list` debug: cpp-stl-forward-list.tex

C++ STL started out with only doubly linked list. Singly linked list was added later. The C++ STL singly linked class is `std::forward_list`.

Run and study the following very carefully:

```
#include <iostream>
#include <string>
#include <forward_list>

void print(std::forward_list< int >::iterator p,
           std::forward_list< int >::iterator q)
{
    std::cout << '[';
    std::string delim = "";
    while (p != q)
    {
        std::cout << delim << (*p); delim = ", ";
        ++p;
    }
    std::cout << "]\n";
}


int main()
{
    std::forward_list< int >::iterator p;
    std::forward_list< int >::iterator q;

    std::cout << "1.\n";
    std::forward_list< int > list;
    std::cout << (list.empty() ? "true" : "false") << '\n';
    // forward_list does not have size
    std::cout << "size: " << std::distance(list.begin(), list.end())
              << '\n';

    // Operations at head/front.
    list.push_front(0);
    list.push_front(1);
    list.push_front(2);
    list.push_front(3);
    print(list.begin(), list.end()); // end() has runtime of O(n)
    list.clear();
    print(list.begin(), list.end());

    std::cout << "2.\n";
    list = {1, 2, 3, 4};
```

```
    print(list.begin(), list.end());
    list.pop_front();
    print(list.begin(), list.end());

    std::cout << "3.\n";
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    list.front() = 42;
    std::cout << list.front() << '\n';
    print(list.begin(), list.end());

    // erase_after an iterator
    std::cout << "4.\n";
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    p = list.begin(); ++p;
    list.erase_after(p);
    print(list.begin(), list.end());

    // erase_after from one iterator to another
    std::cout << "5.\n";
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    p = list.begin(); ++p;
    q = list.end();
    list.erase_after(p, q);
    print(list.begin(), list.end());

    // insert_after an iterator
    std::cout << "6.\n";
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    p = list.begin(); ++p;
    list.insert_after(p, 99999);
    list.insert_after(p, 88888);
    list.insert_after(p, 77777);
    print(list.begin(), list.end());

    // insert_after 5 copies of -1 at iterator
    std::cout << "7.\n";
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    p = list.begin(); ++p;
    list.insert_after(p, 5, -1);
    print(list.begin(), list.end());

    // insert section of a list (specified by two iterators)
    // into another list at an iterator
    //
    // 0---0---0---0---0---0---0---0---0---0
    //             |       |
```

```
    //             V       V
    //             ---------
    //                 V
    //     @---@---@---@---@---@
    //
    //  to get:
    //
    //     @---@---@---@---O---O---O---@---@
    std::cout << "8.\n";
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    p = list.begin(); ++p;
    std::forward_list< int > list1 = {77, 88, 99};
    list.insert_after(p, ++list1.begin(), list1.end());
    print(list.begin(), list.end());

    return 0;
}
```

Here's the output:

```
[student@localhost forward_list] g++ main.cpp; ./a.out
1.
true
size: 0
[3, 2, 1, 0]
[]
2.
[1, 2, 3, 4]
[2, 3, 4]
3.
[1, 2, 3, 4]
42
[42, 2, 3, 4]
4.
[1, 2, 3, 4]
[1, 2, 4]
5.
[1, 2, 3, 4]
[1, 2]
6.
[1, 2, 3, 4]
[1, 2, 77777, 88888, 99999, 3, 4]
7.
[1, 2, 3, 4]
[1, 2, -1, -1, -1, -1, -1, 3, 4]
8.
[1, 2, 3, 4]
[1, 2, 88, 99, 3, 4]
```

The methods are somewhat similiar to C++ STL doubly linked list; see next section.

Exercise 106.9.1. Check out the copy constructor, `operator=`, and `operator==`.

106.10 C++ STL doubly linked list: `std::list` debug:

cpp-stl-list.tex

The C++ STL provides a doubly-linked list class called `std::list`. The following shows you the most basic methods of this class.

Run and study the following very carefully:

```
#include <iostream>
#include <list>         // doubly-linked list

void print(std::list< int >::iterator p,
           std::list< int >::iterator q)
{
    std::cout << '[';
    std::string delim = "";
    while (p != q)
    {
        std::cout << delim << (*p); delim = ", ";
        ++p;
    }
    std::cout << "]\n";
}

int main()
{
    std::list< int >::iterator p;
    std::list< int >::iterator q;

    std::cout << "1.\n";
    std::list< int > list;
    std::cout << (list.empty() ? "true" : "false") << '\n';
    std::cout << "size: " << list.size() << '\n';

    // Operations at head/front.
    list.push_front(0);
    list.push_front(1);
    list.push_back(111);
    list.push_back(222);
    print(list.begin(), list.end());

    std::cout << "2.\n";
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    list.pop_front();
    list.pop_back();
    print(list.begin(), list.end());

    std::cout << "3.\n";
```

```
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    list.front() = 111;
    list.back() = 999;
    print(list.begin(), list.end());

    // erase at an iterator
    std::cout << "4.\n";
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    p = list.begin(); ++p;
    list.erase(p);
    print(list.begin(), list.end());
    p = list.end(); --p;
    print(list.begin(), list.end());

    // erase from one iterator to another
    std::cout << "5.\n";
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    p = list.begin(); ++p;
    q = list.end(); --q;
    list.erase(p, q);
    print(list.begin(), list.end());

    // insert at an iterator (i.e., just before)
    std::cout << "6.\n";
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    p = list.begin(); ++p;
    list.insert(p, 99999);
    list.insert(p, 88888);
    list.insert(p, 77777);
    print(list.begin(), list.end());

    // insert 5 copies of -1 at iterator
    std::cout << "7.\n";
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    p = list.begin(); ++p;
    list.insert(p, 5, -1);
    print(list.begin(), list.end());

    // insert section of a list (specified by two iterators)
    // into another list at an iterator
    //
    // O---O---O---O---O---O---O---O---O---O
    //             |       |
    //             V       V
    //             ---------
    //                 V
```

```
    //     @---@---@---@---@---@
    //
    //  get:
    //
    //     @---@---@---@---O---O---O---@---@

    std::cout << "8.\n";
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    p = list.begin(); ++p;
    std::list< int > list1 = {77, 88, 99};
    list.insert(p, ++list1.begin(), list1.end());
    print(list.begin(), list.end());

    // reverse iterator
    std::cout << "9.\n";
    list = {1, 2, 3, 4};
    print(list.begin(), list.end());
    for (std::list< int >::reverse_iterator p = list.rbegin();
         p != list.rend(); ++p)
    {
        std::cout << (*p) << ' ';
    }
    std::cout << std::endl;

    return 0;
}
```

Here's the output:

```
[student@localhost list] g++ main.cpp; ./a.out
1.
true
size: 0
[1, 0, 111, 222]
2.
[1, 2, 3, 4]
[2, 3]
3.
[1, 2, 3, 4]
[111, 2, 3, 999]
4.
[1, 2, 3, 4]
[1, 3, 4]
[1, 3, 4]
5.
[1, 2, 3, 4]
[1, 4]
6.
[1, 2, 3, 4]
[1, 99999, 88888, 77777, 2, 3, 4]
7.
[1, 2, 3, 4]
[1, -1, -1, -1, -1, -1, 2, 3, 4]
8.
[1, 2, 3, 4]
[1, 88, 99, 2, 3, 4]
9.
[1, 2, 3, 4]
4 3 2 1
```

Of course with `std::list` (since it's a doubly-linked list), you also get a stack, a queue, and a deque. However C++ provides these classes. See next few sections.

Exercise 106.10.1. Recall radix sort using queues (check earlier chapter). Implement LSB radix sort on a vector of integers using an array of queues using `std::queue` (that uses `std::list`). □

106.11 C++ STL deque: `std::deque` debug: cpp-stl-deque.tex

The C++ STL `std::deque` (double ended queue) class is a double-ended queue but the implementation usually uses dynamic arrays, i.e., it's not implemented using a doubly-linked list. Specifically, `std::deque` is implemented as a vector of pointers to vectors of the same size.

Run and study the following very carefully:

```
#include <iostream>
#include <deque>

int main()
{
    std::deque< int > dq;

    dq.push_front(1);
    dq.push_front(2);
    std::cout << dq.front() << ' ' << dq.back() << '\n';

    dq.push_back(100);
    dq.push_back(101);
    std::cout << dq.front() << ' ' << dq.back() << '\n';

    dq.pop_front(); dq.pop_back();
    std::cout << dq.front() << ' ' << dq.back() << '\n';

    std::cout << dq.size() << ' ' << dq.empty() << '\n';

    return 0;
}
```

Here's the output:

```
[student@localhost deque] g++ main.cpp; ./a.out
2 1
2 101
1 100
2 0
```

106.12 C++ STL stack: `std::stack` debug: cpp-stl-stack.tex

The `std::stack` (by default) uses `std::deque` and supports `size()`, `empty()`, `push_back()`, `push()`, `pop_back()`, `pop()`, `back()`, `top()`. Actually `std::stack` is what's called a container adaptor: you can tell specify type of container you want to use for the stack implementation whether it's `std::vector`, `std::list`, or `std::deque`. In fact you can specify *any* class as along as the class supports

- `empty()` and `size()`
- `back()`
- `push_back()` and `pop_back()`

(Think of `back` as `top` of stack.)

Run and study the following very carefully:

```
#include <iostream>
#include <list>
#include <stack>

int main()
{
    // stack using std::deque< int >
    std::stack< int > s1;

    // stack using std::list< int >
    std::stack< int, std::list< int > > s2;

    s1.push(5); s1.push(42);
    s2.push(5); s2.push(42);

    std::cout << s1.top() << ' ' << s2.top() << '\n';
    std::cout << s1.size() << ' ' << s2.size() << '\n';
    std::cout << s1.empty() << ' ' << s2.empty() << '\n';

    s1.pop();
    s2.pop();

    std::cout << s1.top() << ' ' << s2.top() << '\n';
    std::cout << s1.size() << ' ' << s2.size() << '\n';
    std::cout << s1.empty() << ' ' << s2.empty() << '\n';

    return 0;
}
```

Here's the output:

```
[student@localhost stack] g++ main.cpp; ./a.out
42 42
2 2
0 0
5 5
1 1
0 0
```

106.13 C++ STL queue: `std::queue` debug: cpp-stl-queue.tex

There's also a `std::queue` class which is also a container adaptor class like `std::stack`. The class supports

- `empty()` and `size()`
- `front()` and `back()`
- `push()` and `pop()`

(Think of `back` as `tail` and `front` as `head` of queue.)

Run and study the following very carefully:

```
#include <iostream>
#include <list>
#include <queue>

int main()
{
    // queue using std::deque< int >
    std::queue< int > q1;

    // stack using std::list< int >
    std::queue< int, std::list< int > > q2;

    q1.push(5); q1.push(42);
    q2.push(5); q2.push(42);

    std::cout << q1.front() << ' ' << q2.front() << '\n';
    std::cout << q1.back() << ' ' << q2.back() << '\n';
    std::cout << q1.size() << ' ' << q2.size() << '\n';
    std::cout << q1.empty() << ' ' << q2.empty() << '\n';

    q1.pop();
    q2.pop();

    std::cout << q1.front() << ' ' << q2.front() << '\n';
    std::cout << q1.back() << ' ' << q2.back() << '\n';
    std::cout << q1.size() << ' ' << q2.size() << '\n';
    std::cout << q1.empty() << ' ' << q2.empty() << '\n';

    return 0;
}
```

```
[student@localhost queue] g++ main.cpp; ./a.out
5 5
42 42
2 2
0 0
42 42
42 42
1 1
0 0
```

106.14 Applications debug: mathematical-expressions.tex

There are many applications of linked lists (singly or doubly linked), stacks, and queues. (Don't forget that stacks and queues can be implemented using lists so when you use stacks/queues you're might be using linked lists.)

I'm just going to talk about just a couple of them. (Don't be fooled by this short list – linked lists are really very important.)

Linked lists are definitely used in language computation. We have to start somewhere so I'm going to talk about the processing of mathematical languages. Natural language prcoessing (NLP) as in processing human language (example: English) is a lot more complex because the English language is not as precise as the mathematical language. Pure abstract mathematical languages do not have humor, puns, sarcasm, etc.

We'll look at the simplest sentences in mathematical languages. The first few will be on evaluating mathematical expressions, something that you have been doing since elementary school. Besides the standard mathematical expressions such as

```
1 + 2 - 3 * 5
```

we will also look at expressions using a different mathemaical language where a sentence in this language looks like this:

```
+ 3 5
```

The operator is placed *in front*. Then there's this mathematical language where the operator is placed at the back, like this:

```
3 5 +
```

You'll see that the stack and queue can be used as memory devices while you evaluate expressions.

I'll also talk about using linked list to represent common quantities used in computer science and math (example: long integers, matrices).

I'll briefly talk about using linked list to model sparse data structures such as sparse long integers, sparse matrices, and graphs (which are just dots joined by lines) especially when each dot is not joined to too many other dots – again something that's "sparse".

We'll briefly look at some search problems. This is a huge topic. We'll come back to search again later.

But let's start with something that looks naively simple ...

106.15 Reverse debug: reverse.tex

The reverse of a string is just writing the string in the reverse order. For instance the reverse of

`helloworld`

is

`dlrowolleh`

In the case of a string, if the string is stored in `x`, and you want to reverse in `y`, you just do this:

```
for (int i = 0; i < len(s); i++)
{
    y[len(s) - 1 - i] = x[i];
}
y[len(s)] = '\0';
```

In this case the reverse is stored in another array. If you want to store the reverse in the same string, you can copy `y` back to `x`. But if you want to put the reverse back into `x`, you might as well swap the first and last character of `x`, swap the second and second last, etc.

```
int len = strlen(s)
for (int i = 0; i < len / 2; ++i)
{
    x[len - 1 - i] = x[i];
}
```

However there's another way to do this: you put the characters in the string into a stack, and then pop off the values and put is back in the string (or into another if the original shouldn't be destroyed). Here's the code:

```
stack.clear();
for (int i = 0; i < strlen(s); ++i)
{
    stack.push(x[i]);
}
for (int i = 0; i < strlen(s); ++i)
{
    x[i] = stack.top();
    stack.pop();
}
```

So what? Notice that in this new algorithm, you always traverse the string (container of characters) in the same direction, from index 0 to the highest index. In the algorithm above, you (basically) traverse `x` in one direction and `y` in the opposite

direction. You're using the fact that a string can be efficiently traversed in the forward and the backward direction.

Exercise 106.15.1. Singly linked list reversal.

1. Given a singly linked list, replace it with its reverse. Assume the singly linked list is the plain singly linked list, i.e., no sentinel nodes. Assume you have a stack available – this will make it easier. This means that is the singly linked list has values 5, 3, 1, 7 (from head to tail), then after your reverse, the singly linked list has values 7, 1, 5, 3 (from head to tail).
2. Now ... given a singly linked list, reverse the values in the list without using any other container (so not use a stack, array, etc.) Use the least number of extra variables and make sure your runtime is $O(n)$. (This exercise already appear in the section on singly linked list.)

□

Exercise 106.15.2. Suppose you have a singly linked list and I want you to put the values in the reverse order into another singly linked list. Assume you don't have any other container. Of course you can use a constant number of variables. How would you do it? What is the runtime? □

106.16 Palindromes debug: palindromes.tex

A string is a palindrome is it is its own reverse. For instance here's a palindrome:

```
madam
```

Exercise 106.16.1. Write a program that prompts the user for a string and checks if it's a palindrome. Now what if you're only allowed to traverse strings in one direction and you have stack? □

Exercise 106.16.2. (See https://leetcode.com/problems/palindrome-number/.) Write a program that prompts the user for an integer (not string) and checks if it's a palindrome. The integer `5` is a palindrome. So is `12321` and `123321`. However `-1` is not a palindrome (its reverse is `1-`). `1232` is also not a palindrome. Do not convert the integer to a string and do not use an array.

106.17 Balanced expressions debug: balanced-expressions.tex

Another use of the stack is to check for balanced expressions. What do I mean by that? Suppose you look at this arithmetic expression

```
(1 + (2 + 3)) * ((4 - 5 / (7 * 8) + 9) % 0)
```

When you only look at the parentheses you see this:

```
(())((()))
```

When the parentheses come from a well-formed arithmetic expression, we say that the parentheses are balanced. This is not balanced:

```
()(())()(()(()))(())((()))))(((())
```

See it?

Exercise 106.17.1. Write a program that prompts the user for an arithmetic expression and checks if the left and right parentheses form a balanced expression. This should NOT take you more than 5 minutes. □

Now, the problem can be a little bit more complicated: instead of just `(` and `)`, suppose we allow `[` and `]`. How about throwing in `<` and `>`? We might as well include { and } too.

The problem of whether the parentheses in an expression is balanced or not is similar to the problem of checking whether a string is a palindrome. Why? Look at this palindrome again

```
madam
```

You are matching the two `m`'s:

```
madam
```

You then match the `a`'s:

```
madam
```

If you look at this:

```
([()])
```

you see quickly that you're also matching two things in a manner similar to a palindromic check. There's of course a difference because this is also balanced but not palindromic:

```
[()](())
```

Exercise 106.17.2. Design an algorithm that checks if a string containing the characters `(,),{,},<,>,[,]` is balanced. (Hint: Stack. Solution at end of this

subsection.) □

Don't be fooled by the above "puzzles" ... reversing a string, checking palindromes, balancing expressions. Seems like little tricky problems just to tickle the mind. NO! It's a lot more than just meets the eye. The presence of a stack used for a computation indicates that there's a certain theoretical level of complexity in the problem. It's even important and present in nature: palindromic subsequences in DNA in related to the ability of the molecules to perform folding. (Google protein folder.)

Exercise 106.17.3. * This one is much harder than the previous. Write a program that prompts the user for a string containing left and right parentheses and return a balanced expression by making the least changes to the given string, either by deleting or inserting characters. □

Solutions

Solution to Exercise 106.17.2. The idea is to keep the "openings" (, [, <, or { in a stack when scanning the string. And when a "closing"),], >, or } is seen, check is it matches the top of the stack. Here's the pseudocode:

```
Take a character from the string.
If the character is (, [, <, or {, push it onto the stack.
Otherwise,
    If the character is ):
        if the top of the stack is (, pop it off
        if the top of the stack is not, stop with ERROR
    If the character is ]:
        if the top of the stack is [, pop it off
        if the top of the stack is not, stop with ERROR
    If the character is }:
        if the top of the stack is {, pop it off
        if the top of the stack is not, stop with ERROR
    If the character is >:
        if the top of the stack is <, pop it off
        if the top of the stack is not, stop with ERROR
Repeat the above until all characters are processes.
```

106.18 Polish notation

debug: polish-notation.tex

The common mathematical language of arithmetic expression contains a "sentence" of this form:

```
1 + 2 - 3 * 5
```

infix notation

The above way of writing arithmetic expression is called **infix notation**: a binary operator is placed *in the middle*. When given the above, I am interested in evaluating it. Of course I want the algorithm to obey standard precedence rules. You have been using this algorithm since elementary school.

Infix notation requires parentheses. For instance if you look at the following in infix notation

```
1 + 2 * 3
```

it means `1 + (2 * 3)`: the multiplication goes first. If you wanted to perform the `+` first, you are forced to write `(1 + 2) * 3`. Let me repeat the above: if I say to you "Compute this `1 + 2 * 3` but make sure you do multiplication first", then mathematically (without all the above words), I have to write

```
1 + (2 * 3)
```

but if I say "Compute this `1 + 2 * 3` but make sure you do addition first", then mathematically (without all the above words), I have to write

```
(1 + 2) * 3
```

The infix notation requires parenthesis.

prefix notation
Polish notation

Another way to write down an operator with two parameters is to put the operator *in front*. This is called **prefix notation** or **Polish notation** (the Polish mathematician Jan Łukasiewicz invented this notation). For instance instead of writing `3 + 5`, in prefix notation I would write

```
+ 3 5
```

The prefix notation does *not* require parenthesis.

In the prefix notation (or Polish notation), if you want `1 + (2 * 3)` (usual infix) in the prefix/Polish notation, you write

```
+ 1 * 2 3
```

because you evaluate it like this:

```
+ 1 * 2 3
```

i.e., (let me do it slowly):

```
  + 1 * 2 3
= + 1 * 2 3
= + 1 6
= 7
```

If you really want `(1 + 2) * 3`, then you write this in prefix notation

```
* + 1 2 3
```

i.e.,

```
  * + 1 2 3
= * + 1 2 3
= * 3 3
= 9
```

To evaluate an expression in infix notation, note that you can simplify the expression if it's of the form `[operator] [num1] [num2]`. So if you're given a longer expression such as

```
* 1 + 2 - 3 5 4
```

you scan left-to-right, and once you see `[operator] [num1] [num2]`, you simplify. Here's an example where I'm using `|` as my finger and it moves left-to-right, and you can only see the data on the left of `|`:

```
 |* 1 - + 2 - 3 5 4
= *|1 - + 2 - 3 5 4
= * 1|- + 2 - 3 5 4
= * 1 -|+ 2 - 3 5 4
= * 1 - +|2 - 3 5 4
= * 1 - + 2|- 3 5 4
= * 1 - + 2 -|3 5 4
= * 1 - + 2 - 3|5 4
= * 1 - + 2 - 3 5|4
= * 1 - + 2 -2|4       by - 3 5 = -2
= * 1 - 0|4            by + 2 -2 = 0
= * 1 - 0 4|
= * 1 -4|              by - 0 4 = -4
= -4|                  by * 1 -4 = -4
```

The above method is kind of messy as an algorithm. For instance if you look this line:

```
= * 1 - + 2 - 3 5|4
= * 1 - + 2 -2|4        by - 3 5 = -2
```

The only reason we can simplify is because there's a `-` and then two numbers. However your finger is at `5` and you need to know that two steps to the left is an operation. Now *if* we scan *right-to-left*, remembering numbers

```
  * 1 - + 2 - 3 5 4
                ^


  * 1 - + 2 - 3 5 4
              ^


  * 1 - + 2 - 3 5 4
            ^


  * 1 - + 2 - 3 5 4
          ^
```

then on reaching the `-`, we take the last two numbers from our bag and perform subtraction. Of course we the result (i.e., `-2`) back into our bag and continue. The bag contains only numbers if I use this method.

Clearly the bag is organized like this: I need to be able to put numbers into the bag and I need the bag to give me the *last* two numbers. So ... what type of container should we use for this bag of integers? Look at the last sentence again:

> ... I need the bag to give me the *last* two numbers.

So ... you can perform polish notation evaluation you scan right to left and if you use a stack. Note that if we do it this way, the stack only contains integers. For the previous method, the stack contains integers and operators. Here's the computation but using the new method:

```
                      stack (top on left)
  * 1 - + 2 - 3 5 4   []
  * 1 - + 2 - 3 5     [4]
  * 1 - + 2 - 3       [5,  4]
  * 1 - + 2 -         [3,  5, 4]
  * 1 - + 2           [-2, 4]       because - 3 5 is -2
  * 1 - +             [2, -2, 4]
  * 1 -               [0,  4]       because + 2 -2 is 0
  * 1                 [-4]          because - 0 4  is -4
  *                   [1, -4]
                      [-4]          because * 1 -4 is -4
```

In other words, as you scan right to left:

- If the data read is a number, push it on to the stack.
- If the data read is an operator `op`, pop `num1` from stack, pop `num2` from stack, compute `num3 = op num1 num2` and push `num3` onto the stack.
- If there's no more data, the result is the top of the stack (the stack should have only one value).

Of course there's an error in the prefix expression if you have the following situations:

- When the expression is completely processed, the stack has more than one value
- When you read an operator, you cannot pop two values from the stack.

Note that two main ingredients of the above algorithms is

1. You have a stack (for holding numbers).
2. You can perform `op num1 num2`, i.e., the simplest polish notation operation. The `op` is what you see from the input and the `num1` and `num2` are the top two values on the stack.

Exercise 106.18.1. What is the runtime in the above algorithm (in terms of the length of the expression)? □

Exercise 106.18.2. Evaluate the following expressions written in prefix/Polish notation or write ERROR if the expression is invalid.

1. `* + 3 5 - 7 2`
2. `+ / 8 2 - 4 * 7 2`
3. `* 8 * 2 + 3 - 2 / 7 2`
4. `/ + - * 2 3 7 4 2`

□

Exercise 106.18.3. Convert the following expression in infix notation to prefix/Polish notation. (Do not simplify - just convert).

1. `4 + 2 * 3 - 8 / 7`
2. `4 + 2 * (3 - 8) / 7`
3. `(4 + 2 * 3) - 8 / 7`
4. `(4 + 2 * (3 - 8)) / 7`

□

Exercise 106.18.4.

1. Write a wrong expression in polish notation that will result in a stack with 2 numbers at the end of the algorithm.
2. Write a wrong expression in polish notation that will result in a stack with 1 number when an operator is read.

□

Exercise 106.18.5. Implement a polish notation calculator. The input is a vector of strings where the strings represent integers (example `"123"`, `"-321"`) or binary integer operators, i.e. `"+"`, `"-"`, `"*"`, `"/"`, `"%"`. The return value is an integer. (Besides the algorithm above, clearly you will need to be able to convert a numeric string to an integer. Check your CISS240 notes.) If you do not have enough values in the stack, throw the exception `StackUnderflowError`. If you have too many values in the stack, throw the exception `StackOverflowError`. □

106.19 Postfix notation; reverse Polish notation (RPN) debug: rpn.tex

postfix notation

reverse Polish notation

There is an analogous **postfix notation** or **reverse Polish notation** (RPN). Instead of our usual `3 + 5` (using infix notation), of `+ 3 5` (using prefix notation or Polish notation), in the postfix notation or the reverse Polish notation (RPN), you would write

```
3 5 +
```

Here's the complete evaluation of an RPN expression:

```
  1 2 3 5 + - 4 - *
= 1 2 8 - 4 - *       by 3 5 + = 8
= 1 -6 4 - *          by 2 8 - = -6
= 1 -10 *             by -6 4 - = -10
= -10
```

Here's how to do it in a more systematic way: Think of `|` as your finger in the following computation. Your finger scans left to right. In other words, you can only *see* what's to the left of your finger. Once you see `[num1] [num2] [op]` you can simplify. Here we go ...

```
  |1 2 3 5 + - 4 - *
=  1|2 3 5 + - 4 - *
=  1 2|3 5 + - 4 - *
=  1 2 3|5 + - 4 - *
=  1 2 3 5|+ - 4 - *
=  1 2 3 5 +|- 4 - *
=  1 2 8|- 4 - *          by 3 5 + = 8
=  1 2 8 -|4 - *
=  1 -6|4 - *             by 2 8 - = -6
=  1 -6 4|- *
=  1 -6 4 -|*
=  1 -10|*                by -6 4 - = -10
=  1 -10 *|
=  -10|                   by 1 -10 * = -10
```

Make sure you study the above carefully. Once you get the hang of the above computation, the following should not be surprising:

```
stack (top on right)
[]                  1 2 3 5 + - 4 - *
[1]                   2 3 5 + - 4 - *
[1, 2]                  3 5 + - 4 - *
[1, 2, 3]                 5 + - 4 - *
[1, 2, 3, 5]                + - 4 - *
[1, 2, 8]                     - 4 - *   because 3 5 + is 8
[1, -6]                         4 - *   because 2 8 - is -6
[1, -10]                            *   because -6 4 - is -10
[-10]                                   because 1 -10 * is -10
```

```
EVALUATE-RPN
INPUT: x (a list of integers and operators)
OUTPUT: r (an integer)

let s be a stack
repeat the following until s is empty:
    remove a symbol from s: call it y
    if y is an integer push that onto s
    otherwise y is an operator and do the following:
        pop num2 from s
        pop num1 from s
        compute (num1 y num2) and push the value onto s
pop r from s
if s is empty, return r
otherwise ERROR
```

In the above `compute (num1 y num2)` simply means evaluating the operator `y`. For instance `compute (5 + 3)` gives you 8. □

Exercise 106.19.1. What is the runtime in the above algorithm (in terms of the length of the expression)? □

Exercise 106.19.2. Evaluate the following RPN expressions or write ERROR is the expressions are invalid.

1. `1 3 2 * + 4 5 * -`
2. `3 * 2 / 2 - 4 + 5`

□

Exercise 106.19.3. Convert the following expression in infix notation to postfix/RPN notation. (Do not simplify - just convert).

1. `4 + 2 * 3 - 8 / 7`
2. `4 + 2 * (3 - 8) / 7`
3. `(4 + 2 * 3) - 8 / 7`
4. `(4 + 2 * (3 - 8)) / 7`

□

106.20 Infix notation debug: infix.tex

The "usual" way of writing arithmetic expression where a binary operator is placed in the middle is called infix notation. Here's an example

```
1 + 2 * 3
```

Now as mentioned earlier, you need parentheses if you want to change the standard order of evaluating as infix expression. So any algorithm to evaluate infix expressions must taken into account parentheses – which is a pain. For instance

```
(1 + 2) * 3
```

Of course you can do this: make one scan of the string processing only the highest order operators. Then I redo that with the operators at the second precedence level. Etc. For instance, say your algorithm can handle only `+,-,*,/` then for the following you first do `*,/` scanning the string once, and then you do `+,-` scanning the resulting string a second time:

```
  0 - 1 + 2 * 3 - 4 / 5 * 6
= 0 - 1 + 6 - 4 / 5 * 6
= 0 - 1 + 6 - 0 * 6
= 0 - 1 + 6 - 0
= -1 + 6 - 0
= 5 - 0
= 5
```

(The underline denotes my finger running across the string left-to-right and stopping when I see an operator that I should evaluate.)

Exercise 106.20.1. Implement the above idea. What is the runtime in terms of the length of the string? (Assume that all integers are 1-digit in length.) Assume that the string is given as a queue (the left end is the front of the queue). What data structures do you need? □

But let's see if we can do better ...

The general idea is actually pretty simple. For now, I'll use the idea below to compute arithmetic expressions. Later on, we'll see that it can be used to build "trees".

Suppose you look at

```
1 + 2 * 3
```

How would you evaluate it as a program? How do you decide that `+` should come after `*`? Of course as you process the character left to right, you can't see `*` yet if you're processing `+` or even when you're processing `2`. You would then have to remember `1 + 2` and when you process `*`, you know that you cannot process `1 + 2` because `2` should be used by `*`: 2 *belongs* to `*`. So we have to keep `1 + 2` somewhere.

Now suppose we know that we need to process `2 * 3` first. (This is due to historical conventions – we do multiplications before additions.) What does that mean? We have to remember `1 + 2`, but don't execute the addition. We then have to take the `2` out of what we remembered for executing `2 * 3`. That means that we're taking the *last* thing out of our memory device. Right? The `1` went in first, then the `+` and then the `2` – `2` was the *last* thing we have to remember.

See that?

We have to take `2` out of the things we stash somewhere (the `1` and the `+` and the `2`) and then we took the `2` out. Again this is important: `2` was the last thing we have to remember and `2` is the thing we have to take out from our stash.

AHA! ... We probably need a stack!

In the above case, how do we know that `*` can go?

Because there's no more operators to compete against it. And why must `+` wait for `*`? Because `*` has a higher precedence. To make it easy to read, I'm going to write

```
* > +
```

to say that `*` has higher precedence. Now why is there no one else to compete again `*`? Because it's the last operator – there's no operator after it. Note that this can happen in a different way too ... look at

```
1 + 2 * 3 * 4
```

The first `*`, i.e.,

```
1 + 2 * 3 * 4
```

beat the second `*`

```
1 + 2 * 3 * 4
```

because they are at the *same* precedence level ... and `*` is **left-to-right associative** i.e.,

left-to-right associative

```
1 * 2 * 3
```

means

```
(1 * 2) * 3
```

This behavior of `*` is also called **left associative**.

left associative

So the first * beats the second. Remember this!

But what if I have

```
1 + 2 * 3 + 4
```

and I'm now about to process the second +:

```
1 + 2 * 3 + 4
```

The subexpression

```
1 + 2 * 3
```

is not processed yet. Why? Because the * beats the + so + has to wait. But I also cannot let * go ahead since the next operator might beat *. Now when I read the + in

```
1 + 2 * 3 + 4
```

I know now that * can go ahead. This gives me

```
1 + 6 + 4
```

But wait! ... Since + is left-to-right associative, this means that the first + also go ahead too so that I get this:

```
7 + 4
```

So when do I stop in the general case? From the above example, it's clear that I keep going until I see an operator with precedence order strictly less than the + that I just read. Correct? Remember this!

So ... hmmm ... what if we have this:

```
1 * 2 + 3
```

Suppose we've read `1 * 2` and stored them somewhere (but it's not evaluvated yet ... we're waiting to see if the next operator should go first) and I'm about to the +. Then once we see the +, since * beats +,

```
* > +
```

we know that * can go first. So we pull `1 * 2` out of our memory so as to evaluate it. That gives me 2. Now I have to do the +, i.e.., I have to process `2 + 3`.

In order to use previous processing rules, I will first put the 2 (due to the evaluation of `1 * 2`) back into my memory device. This will make the process uniform. For instance with 2 in the memory device, after processing `+ 3`, I will have `2 + 3` in the memory device. Only when I see that there's no more input for processing then I know that no other operator is beating + and therefore I can compute `2 + 3`.

Wait wait wait ... that means that when I look at the +, I need to recall the `1 * 2`

from a container but the important value is `*`. So if I store `1 * 2` in a stack, the `*` is *not* at the top. Sure, I can pop `2` off and then I will see the `*`.

AHA! ... So ...

For me to see the `*` quickly, I can keep the operators in a separate memory device. So I think I'll have *two* memory devices: one for the integers and one for the operators.

What about something like

```
1 + 2 + 3
```

In that case, once I've remember `1 + 2`, when I see the next `+` I know that `+` remembered can go ahead since they are the same precedence level and `+` associates left to right. Right?

OK. We now have enough information to do a complete trace using the above ideas.

- We continually read the data from the input. The input is made up of integers and operators.
- For memory, we have two stacks, the first for remembering integers (call it `intstack`) and second for operators (call it `opstack`).
- When I read an integer, I push it onto `intstack`.
- When I read an operator `op`, I do the following:
  - If `op` $>$ top of `opstack`: I push `op` onto `opstack`.
  - If `op` $\leq$ top of `opstack`: I pop an operator off `opstack` and use it to evaluate the top two values on `intstack`, pushing the result back onto `intstack`. I keep doing this until `op` $>$ the top of `opstack`. After that I push `op` onto `opstack`.
- When there's no more input to read, I continually pop an operator off `opstack` and evaluate two values from the top of `intstack`, pushing the result back onto `intstack`. The final result is on the top of `intstack`.

Clearly the two cases of processing the `op` can be combined. So I get this:

- We continually read the data from the input. The input is made up of integers and operators.
- For memory, we have two stacks, the first for remembering integers (call it `intstack`) and second for operators (call it `opstack`).
- When I read an integer, I push it onto `intstack`.
- When I read an operator `op`, I do the following:
  - As long as `op` $\leq$ top of `opstack`: I pop an operator off `opstack` and use it to evaluate the top two values on `intstack`, pushing the result back onto `intstack`.
  - After that, I push `op` onto `opstack`.
- When there's no more input to read, I continually pop an operator off `opstack` and evaluate two values from the top of `intstack`, pushing the result back

onto `intstack`. The final result is on the top of `intstack`.

(You see that there's a lot of "operate on the top two values on the stack and put the result back onto the top". See the next section for **stack machines**.)

There are some optimizations: For instance on evaluating with an operator on the `opstack`, note that I pop two values off the `intstack`, operator on them, push the result back onto `intstack`. I keep doing this. So it's better *not* to push the result back onto the stack since it will be popped off again in the next around. You push the result only when the loop has ended. So the above becomes this:

- We continually read the data from the input. The input is made up of integers and operators.
- For memory, we have two stacks, the first for remembering integers (call it `intstack`) and second for operators (call it `opstack`).
- When I read an integer, I push it onto `intstack`.
- When I read an operator `op`, I do the following:
  - If `op` $\leq$ top of `opstack`:
    - I pop off an integer from the `intstack` and store it in `a`
    - As long as `op` $\leq$ top of `opstack`: I pop an operator off `opstack`, pop off an integer from the `intstack` and store it in `b`. I compute `(b op a)` and store it in `a`.
    - I push `a` onto `intstack`
  - After that, I push `op` onto `opstack`.
- When there's no input, I do the following: If `opstack` is empty, I return the top of `intstack`. Otherwise I do the following:
  - I pop an integer off `intstack` and store in `a`.
  - As long as `opstack` is not empty, I do the following: I pop an operator off `opstack` and store in `op`, I pop an integer off `intstack` and store in `b`, I compute `(b op a)` and store in `a`.
  - Otherwise, I pop an integer off the top of `intstack` and store in `a`. (If there are more than two values in `intstack`, there's an error.)

Once we're comfortable with the ideas, we'll write down the algorithm. Let's forget about the parentheses for now and try an example using what we have discovered. Let's look at this:

```
0 - 1 + 2 * 3 - 4 / 5 * 6
```

You would expect the evaluation to be like this:

```
((0 - 1) + (2 * 3)) - ((4 / 5) * 6)
```

Here we go!

```
              intstack opstack (top on left)
                                PROCESS INPUT
0-1+2*3-4/5*6 []        []
                                0:Push intstack
 -1+2*3-4/5*6 [0]       []
                                -:Push opstack
  1+2*3-4/5*6 [0]       [-]
                                1:Push intstack
   +2*3-4/5*6 [0,1]     [-]
                                +:+ <= -. Compute with top of
                                  opstack until opstack empty or
                                  + > top.
                                  Then push + onto opstack.
   +2*3-4/5*6 [-1]      []
   +2*3-4/5*6 [-1]      [-]
    2*3-4/5*6 [-1]      [+]
                                2:Push intstack
     *3-4/5*6 [-1,2]    [+]
                                *:* > +. Push opstack
      3-4/5*6 [-1,2]    [+,*]
                                3:Push intstack
       -4/5*6 [-1,2,3]  [+,*]
                                -:- <= *. Compute with top of
                                  opstack until opstack empty or
                                  - > top.
                                  Then push - onto opstack
        4/5*6 [-1,6]    [+]
        4/5*6 [5]       []
        4/5*6 [5]       [-]
                                4:Push intstack
         /5*6 [5,4]     [-]
                                /:/ > -. Push opstack
          5*6 [5,4]     [-,/]
                                5:Push intstack
           *6 [5,4,5]   [-,/]
                                *:* <= /. Compute with top of
                                  opstack until opstack empty or
                                  * > top.
                                  Then push * onto opstack
            6 [5,4,5]   [-,/]
            6 [5,0]     [-]
                                6:Push intstack
              [5,0,6]   [-,*]
                                NO MORE INPUT: Compute with
                                top of opstack until opstack
                                empty
              [5,0]     [-]
              [5]       []
```

You can see the computations with this:

0 - 1 + 2 * 3 - 4 / 5 * 6

The algorithm at this point look like this:

```
ALGORITHM: EVALUATE-INFIX (without parentheses)
INPUT: x = list/string of integers and operators

intstack is a stack of integers
opstack is a stack of operators

while there is still data in x:
    take data out of x
    if data taken out of x is an int:
        push that int onto intstack
    otherwise (i.e., data is an operator)
        call the data op
        if opstack is not empty and op <= top of opstack:
            let a = pop intstack
            while opstack is not empty and
                op <= top of opstack:
                let b = pop intstack
                let op1 = pop opstack
                let a = (b op1 a)
            push a onto intstack
        push op onto opstack

if opstack is empty:
    return top of intstack
else:
    let a = pop intstack
    while opstack is not empty:
        let b = pop intstack
        let op1 = pop opstack
        let a = (b op1 a)
    return a
```

In the above, when I write "`(b op1 a)`" I mean evaluate the operator `op1` with values `b` and `a`. For instance is `op1` is `-` and `a` and `b` are `2` and `7` respectively, then `(b op a)` is `(7 - 2)` i.e., `5`.

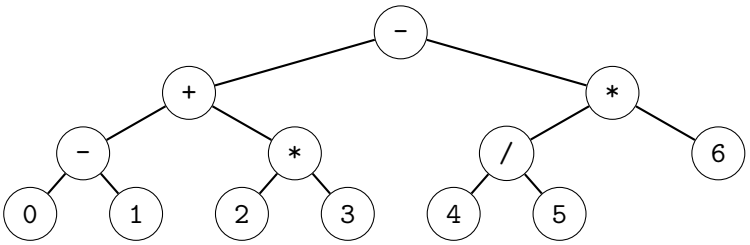
You can add pseudocode to return error codes. For instance just before returning, the stack should have exactly one value – you can check for that. Also, remember that since all operators considered here are binary, you need two integers on the stack to evaluate the operator. You can check for that; if you don't have two integers, then again you can return an error code.

One very important observation is this: Although `*` and `/` has to go first because

they have higher precedence order, note that in the above the leftmost subtraction `-` actually gets evaluated first. But this is OK since the end result is the same. So this algorithm obeys operator precedence rules *and* it evaluates as soon as it is possible. If you look at the above picture:

```
0 - 1 + 2 * 3 - 4 / 5 * 6
```

you'll see that the subtract on the left will not "disturb" the multiplication in the middle. I haven't talked about trees yet, but if you look at this expression tree, you should be able to guess what's happening:



Think of computation with this tree as "bottom to top". You'll notice that the subtree corresponding to the subtraction on the left and multiplication in the middle (of the original expression) are in different subtrees that do not disturb each other. So although `*` should go before the leftmost `-`, the result would be unchanged if I do the leftmost `-` first. And why is "evaluate as soon as possible" a good thing?

Because this means that your containers will remove contents sooner and therefore you save on space.

Exercise 106.20.2. Here's another important observation: Do you notice that in the above trace, the operators on the operator stack is always in decreasing precedence order as you go from the top of the stack to the bottom? Is this always true? □

Exercise 106.20.3. Using the ideas from above (and after studying the trace above), now trace the following yourself:

```
1 - 5 * 3 / 2 + 1 + 2 * 3 % 4
```

What is the maximum stack size? (Try a few more until you are familiar with the algorithm.) □

Next let's handle `(` and `)`. What should we do if we see

```
1 * (2 + 3)
```

Say I have processed `1 *`. What should happen to them? Of course there's no evaluation: I remember these two pieces of data, the integer in one stack and the multiplication in another. Now, when I see the left parenthesis:

```
1 * (2 + 3)
```

if you do this by hand, you wouldn't do any computation. The left parenthesis is just to force us to work on this subexpression:

```
1 * (2 + 3)
```

Now ... this does mean that in some sense the `(` *blocks* us from processing `1 *`. So ... if we put the `(` onto the stack of operators, all we need to do is to make sure our algorithm does not let us past `(`, except when we see a `)`. When you look at previous ideas, all we need to do is to make sure that `+,-,*,/,%` have higher precedence than `(`.

And what do we do when we see `)`? That means that the subexpressions enclosed by this `)` (and the matching `(` that started this subexpression) must be completely evaluated.

To be concrete consider this:

```
1 + (2 + 3 * 4 - 5) / 6
```

Say we have read the data up to this point:

```
1 + (2 + 3 * 4 - 5) / 6
```

On reading `(`, I will push it onto my operator stack. Then I proceed using previous ideas so that `2` is pushed onto the integer stack. Now when I read the `+`:

```
1 + (2 + 3 * 4 - 5) / 6
```

since I impose the rule that `(` has lower precedence, `(` will stay on the stack; `+` is pushed onto the operator, waiting to see if it beats the next operator.

After `3` is pushed onto the integer stack, I read `*`:

```
1 + (2 + 3 * 4 - 5) / 6
```

`*` beats `+` (on the top of the operator stack). So `*` is pushed onto the operator stack and waits to see if it beats the next operator. After `4` is pushed onto the integer stack, I see `-`:

```
1 + (2 + 3 * 4 - 5) / 6
```

Ahhh ... at this point `*` on top of the operator stack beats `-`, so multiplication goes ahead and using `3` and `4` on the integer stack with `*`, I get `12` which is pushed back

onto the integer stack. At this point the operator stack contains

```
[+, (, +]
```

(top of the stack is on the right) and the integer stack contains

```
[1, 2, 12]
```

But `-` is `<=` `+` the top of the operator stack (and they are both left-to-right associative). So `+` gets to go ahead. Which means that the operator stack becomes

```
[+, (]
```

and the integer stack becomes

```
[1, 14]
```

At this point, since `(` < `-`, I push `-` onto the operator stack. `5` is then read and stored on the integer stack:

```
1 + (2 + 3 * 4 - 5) / 6
```

And now ... here's where things get interesting. I read the `)`:

```
1 + (2 + 3 * 4 - 5) / 6
```

I now operate on all the operators on the operator stack *until I see a matching* `(`. The operator stack looks like this:

```
[+, (, -]
```

(don't forget that the multiplication and addition has already gone ahead and are evaluated) and the integer stack looks like this:

```
[1, 14, 5]
```

(The `14` comes from `2 + 3 * 4`.) This means that I will be performing `14 - 5` (to get `9`). After that, I see `(` on the stack. I stop the operations, throw away the `(`. I do *not* push the `)` onto the stack. At this point the original subexpression `(2 + 3 * 4 - 5)` is evaluated (to get `9`) and is placed on the stack. In other words the integer stack looks like this:

```
[1, 9]
```

and the operator stack looks like this:

```
[+]
```

Get it?

Now you might be worried about this point ... I said that once I see `)`, it signals

that it's time to completely finish the subexpression. Then I said to work through all the operators on the stack until I see the first `(` to appear on the top of the operator stack. Notice that I don't check for precedence – I simply go through the operators on the operator stack one at a time until I see `(`. Why don't I worry about the precedence order in this case? Because ... after the `(`, remember that if I see an operator with strictly higher precedence I push, and if I see one that is lower or equal precedence, I evaluate the opereator that is already on the top of the stack. Think about this:

That means that the operators after `(` all the way to the top of the stack must be in ascending precedence order!!!! For instance if I'm reading the following the expression at `)`:

```
1 + (2 + 3 * 4) + 5
```

the operator stack looks like this:

```
[+, (, +, *]
```

It's impossible to have this scenario for your operator stack when you hit a `)`:

```
[+, (, *, -]
```

and it's also impossible to have this:

```
[+, (, +, -]
```

It's also possible to have this:

```
[+, (, +, *, (, +, /, (, -, %]
```

i.e., from the `(` up to (but not including) the next `(`, the precedence order increases. Again, the operator from the `(` to the top (or up the next `(` but not including it) must be in ascending precedence order:

```
[+, (, +, *, (, +, /, (, -, %]
[+, (, +, *, (, +, /, (, -, %]
```

So when I evaluate starting at the top of the operator stack down to `(`, I am going from high to low precedence order. That's why I don't have to check for precedence order for this scenario. Get it?

Here's a trace:

```
                 intstack opstack (top on left)
                                                PROCESS INPUT
0-(1+2+3*(4-5)+6) []         []
                                                0:Push intstack
 -(1+2+3*(4-5)+6) [0]        []
                                                -:Push opstack
  (1+2+3*(4-5)+6) [0]        [-]
                                                (:Push opstack
   1+2+3*(4-5)+6) [0]        [-,(]
                                                1:Push intstack
    +2+3*(4-5)+6) [0,1]      [-,(]
                                                +:+ > (. Push opstack
     2+3*(4-5)+6) [0,1]      [-,(,+]
                                                2:Push intstack
      +3*(4-5)+6) [0,1,2]    [-,(,+]
                                                +:+ <= +. Compute with top of
                                                  opstack until opstack empty or
                                                  + > top.
                                                  Then push + onto opstack.
       3*(4-5)+6) [0,3]      [-,(]
       3*(4-5)+6) [0,3]      [-,(,+]
                                                3:Push intstack
        *(4-5)+6) [0,3,3]    [-,(,+]
                                                *:* > +. Push opstack
         (4-5)+6) [0,3,3]    [-,(,+,*]
                                                (:Push opstack
          4-5)+6) [0,3,3]    [-,(,+,*,(]
                                                4:Push intstack
           -5)+6) [0,3,3,4]  [-,(,+,*,(]
                                                -:- > (. Push opstack
            5)+6) [0,3,3,4]  [-,(,+,*,(,-]
                                                5:Push intstack
             )+6) [0,3,3,4,5][-,(,+,*,(,-]
                                                ):Compute with top of opstack
                                                  until (.
              +6) [0,3,3,-1][-,(,+,*,(]
              +6) [0,3,3,-1][-,(,+,*]
                                               +:+ <= *. Compute with top of
                                                 opstack until opstack empty or
                                                 + > top.
                                                 Then push + onto opstack.
               6) [0,3,-3]  [-,(,+]
               6) [0,0]     [-,(]
               6) [0,0]     [-,(,+]
               6) [0,0]     [-,(,+]
                                               6:Push intstack
                ) [0,0,6]   [-,(,+]
                                               ):Compute with top of opstack
                                                  until (.
                  [0,6]     [-,(]
                  [0,6]     [-]
                                               NO MORE INPUT: Compute with
                                               top of opstack until opstack
                                               empty
```

```
ALGORITHM: EVALUATE-INFIX (with parentheses)
INPUT: x = list/string of integers and operators

intstack is a stack of integers
opstack is a stack of operators

while there is still data in x:
    take data out of x
    if data taken out of x is an int:
        push that int onto intstack
    otherwise (i.e., data is an operator)
        call the data op
        if op is '(':
            push op onto opstack
        elif op is ')':
            let a = pop intstack
            while top of opstack is not '(':
                let b = pop intstack
                let op1 = pop opstack
                let a = (b op1 a)
            push a onto intstack
        otherwise:
            if opstack is not empty and op <= top of opstack:
                let a = pop intstack
                while opstack is not empty and
                    op <= top of opstack:
                    let b = pop intstack
                    let op1 = pop opstack
                    let a = (b op1 a)
                push a onto intstack
            push op onto opstack

if opstack is empty:
    return top of intstack
else:
    let a = pop intstack
    while opstack is not empty:
        let b = pop intstack
        let op1 = pop opstack
        let a = (b op1 a)
    return a
```

Exercise 106.20.4. What if you do this: Ensure that all subexpressions are parenthesized – include the largest one. So if you need to evaluate `1 + 2 * 3`, evaluates

this instead: `(1 + 2 * 3)`. How will that change your pseudocode? □

Exercise 106.20.5. Write a C++ function that accepts an infix expression described by a `std::vector` of strings and perform infix evaluation on the expression. For instance when you execute this

```
// 0-(1+2+3*(4-5)+6)
std::vector< std::string> > e;
e.push_back("0");
e.push_back("-");
e.push_back("(");
e.push_back("1");
e.push_back("+");
e.push_back("2");
e.push_back("+");
e.push_back("3");
e.push_back("*");
e.push_back("4");
e.push_back("-");
e.push_back("5");
e.push_back(")");
e.push_back("+");
e.push_back("6");
int x = eval_infix(s, true);
```

The integer value of `-6` is given to `x` and the following output is shown below. (If the last parameter is set to `false` then `eval_infix()` will not print any output.)

```
0-(1+2+3*(4-5)+6) []          []
 -(1+2+3*(4-5)+6) [0]         []
  (1+2+3*(4-5)+6) [0]         [-]
   1+2+3*(4-5)+6) [0]         [-,(]
    +2+3*(4-5)+6) [0,1]       [-,(]
     2+3*(4-5)+6) [0,1]       [-,(,+]
      +3*(4-5)+6) [0,1,2]     [-,(,+]
       3*(4-5)+6) [0,3]       [-,(]
       3*(4-5)+6) [0,3]       [-,(,+]
        *(4-5)+6) [0,3,3]     [-,(,+]
         (4-5)+6) [0,3,3]     [-,(,+,*]
          4-5)+6) [0,3,3]     [-,(,+,*,(]
           -5)+6) [0,3,3,4]   [-,(,+,*,(]
            5)+6) [0,3,3,4]   [-,(,+,*,(,-]
             )+6) [0,3,3,4,5] [-,(,+,*,(,-]
              +6) [0,3,3,-1]  [-,(,+,*,(]
              +6) [0,3,3,-1]  [-,(,+,*]
               6) [0,3,-3]    [-,(,+]
               6) [0,0]       [-,(]
```

```
6) [0,0]        [-,(,+]
6) [0,0]        [-,(,+]
 ) [0,0,6]      [-,(,+]
   [0,6]        [-,(]
   [0,6]        [-]
   [-6]         []
```

[Put the output into array of strings and then figure out the maximum column width that you need when everything is done. *Then* you perform the printing.] □

Exercise 106.20.6. Using the ideas from above (and after studying the trace above), now trace the following yourself:

```
1 - 5 * (3 / 2) + (1 + (2 * 3) % 4)
```

Try a few more until you're comfortable with the algorithm. □

Exercise 106.20.7. Think about this very carefully: all the operators in the above examples are left-to-right associative, i.e. left associative. For instance $1+2+3+4$ means $((1+2)+3)+4$. This is the same for subtraction, integer division, and integer mod. What if an operator is right associative? Suppose you also want to handle exponentiation. Say we use the character `^` to denote exponentiation. For instance `2 ^ 3` is `8`. Recall that `^` is right-to-left associative, i.e., right associative. In other words `2 ^ 3 ^ 4 ^ 5` is `2 ^ (3 ^ (4 ^ 5))`. This is different from `+` which is left-to-right associative: `2 + 3 + 4` is `(2 + 3) + 4)`. □

Exercise 106.20.8. The above examples are binary operators. What about unary operators? You know that `+` can be either a binary operator or a unary operator. The unary `+` is the "positive of" operator. To avoid this ambiguity, let's talk about the factorial. How about adding the factorial to the above algorithm? For instance `3!` is `6`. Note that it's unary. Of course `3!!!` is `((3!)!)!`. After you are done think about this: `!` is written on the *right* of the argument. The "positive of" `+` and "negative of" `-` is written on the *left* of the argument. How does the computation of `++++3` differ from `3!!!!`? □

106.21 Infix to RPN debug: infix-to-rpn.tex

Now, instead of evaluating an infix expression, you can also convert an infix expression to a postfix/RPN that evaluates to the same value. What I mean is this: Suppose you're given

```
1 + 2
```

Your program give me this:

```
1 2 +
```

The infix expression for the first trace in the previous section is

```
0 - 1 + 2 * 3 - 4 / 5 * 6
```

We expected the evaluation to follow this:

```
(((0 - 1) + (2 * 3)) - ((4 / 5) * 6))
```

Now if I do this carefully, the postfix/RPN expression should be this:

```
  (((0 - 1) + (2 * 3)) - ((4 / 5) * 6))
= (((0 1 -) + (2 * 3)) - ((4 / 5) * 6))
= (((0 1 -) + (2 3 *)) - ((4 / 5) * 6))
= (((0 1 -) (2 3 *) +) - ((4 / 5) * 6))
= (((0 1 -) (2 3 *) +) - ((4 5 /) * 6))
= (((0 1 -) (2 3 *) +) - ((4 5 /) 6 *))
= (((0 1 -) (2 3 *) +) ((4 5 /) 6 *) -)
```

(Yes, yes, I know you probably shouldn't mix infix and postfix/RPN and in fact you probably won't find the above trick in books. But clearly the above helps.) So the corresponding postfix/RPN expression that evaluates in the same way is

```
0 1 - 2 3 * + 4 5 / 6 * -
```

Here's the trace of the above infix expression from the previous section. Let's see if we can derive some inspiration from it.

```
              intstack opstack (top on left)
                                PROCESS INPUT
0-1+2*3-4/5*6 []        []
                                0:Push intstack
 -1+2*3-4/5*6 [0]       []
                                -:Push opstack
  1+2*3-4/5*6 [0]       [-]
                                1:Push intstack
   +2*3-4/5*6 [0,1]     [-]
                                +:+ <= -. Compute with top of
                                  opstack until opstack empty or
                                  + > top.
                                  Then push + onto opstack.
   +2*3-4/5*6 [-1]      []
   +2*3-4/5*6 [-1]      [-]
    2*3-4/5*6 [-1]      [+]
                                2:Push intstack
     *3-4/5*6 [-1,2]    [+]
                                *:* > +. Push opstack
      3-4/5*6 [-1,2]    [+,*]
                                3:Push intstack
       -4/5*6 [-1,2,3]  [+,*]
                                -:- <= *. Compute with top of
                                  opstack until opstack empty or
                                  - > top.
                                  Then push - onto stack2.
        4/5*6 [-1,6]    [+]
        4/5*6 [5]       []
        4/5*6 [5]       [-]
                                4:Push intstack
         /5*6 [5,4]     [-]
                                /:/ > -. Push opstack
          5*6 [5,4]     [-,/]
                                5:Push intstack
           *6 [5,4,5]   [-,/]
                                *:* <= /. Compute with top of
                                  opstack until opstack empty or
                                  * > top.
                                  Then push * onto opstack
            6 [5,4,5]   [-,/]   .
            6 [5,0]     [-]
                                6:Push intstack
              [5,0,6]   [-,*]
                                NO MORE INPUT: Compute with
                                top of opstack until opstack
                                empty
              [5,0]     [-]
              [5]       []
```

AHA! Notice that for the postfix/RPN expression

```
0 1 - 2 3 * + 4 5 / 6 * -
```

The first part is

```
0 1 - ...
```

and in the trace above, you see it here:

```
              intstack opstack (top on left)
                               PROCESS INPUT
0-1+2*3-4/5*6 []       []
                               0:Push intstack
 -1+2*3-4/5*6 [0]      []
                               -:Push opstack
  1+2*3-4/5*6 [0]      [-]
                               1:Push intstack
   +2*3-4/5*6 [0,1]    [-]
                               +:+ <= -. Compute with top of
                                 opstack until opstack empty or
                                 + > top.
                                 Then push + onto opstack.
   +2*3-4/5*6 [-1]     []
...
```

The `-1` in the integer stack at the last line is of course due to `0 - 1` where the `0` and the `1` and the `-` are from the integer stack and the operator stack from the next step. Now in the case of getting the postdfix/RPN expression, I want `0 1 -` ... so instead of evaluating, I just extract those values and put them into a container. Right? The trace becomes:

```
              intstack opstack (top on left)
                               PROCESS INPUT
0-1+2*3-4/5*6 []       []
                               0:Push intstack
 -1+2*3-4/5*6 [0]      []
                               -:Push opstack
  1+2*3-4/5*6 [0]      [-]
                               1:Push intstack
   +2*3-4/5*6 [0,1]    [-]
                               +:+ <= -. Compute with top of
                                 opstack until opstack empty or
                                 + > top.
                                 Then push + onto opstack.
   +2*3-4/5*6 []       []
              [0,1,-]
...
```

The next part of the postfix/RPN

```
0 1 - 2 3 * + 4 5 / 6 * -
```

is this:

```
... 2 3 * ...
```

Again, it appears in the above trace:

```
              intstack opstack (top on left)
                               PROCESS INPUT
0-1+2*3-4/5*6 []       []
                               0:Push intstack
 -1+2*3-4/5*6 [0]      []
                               -:Push opstack
  1+2*3-4/5*6 [0]      [-]
                               1:Push intstack
   +2*3-4/5*6 [0,1]    [-]
                               +:+ <= -. Compute with top of
                                 opstack until opstack empty or
                                 + > top.
                                 Then push + onto opstack.
   +2*3-4/5*6 [-1]     []
   +2*3-4/5*6 [-1]     [-]
    2*3-4/5*6 [-1]     [+]
                               2:Push intstack
     *3-4/5*6 [-1,2]   [+]
                               *:* > +. Push opstack
      3-4/5*6 [-1,2]   [+,*]
                               3:Push intstack
       -4/5*6 [-1,2,3] [+,*]
                               -:- <= *. Compute with top of
                                 opstack until opstack empty or
                                 - > top.
                                 Then push - onto stack2.
        4/5*6 [-1,6]   [+]
...
```

The 6 is of course due to 2 * 3.

Now ... if you think about it, the integer stack at the last line of the trace

```
...
        4/5*6 [-1,6]   [+]
...
```

If I had kept the [0,1,-] there

```
...
        4/5*6 [0,1,-,6]    [+]
...
```

and likewise, instead of computing `2 * 3` to get `6`, I leave the `2 3 *` in that container, I would have

```
...
        4/5*6 [0,1,-,2,3,*]    [+]
...
```

With more experiments, you'll realize that the same algorithm that evaluates an infix expression can be easily used to convert an infix expression to the corresponding postfix/RPN expression. The only difference is that in the evaluation function, you operate with the operator popped from the operator stack together with two integers from the integer stack. Since we're not operating on the integers, we don't have to pop them. The operator is then just pushed on top of the integer stack. Which by the way, we should not be called the first data structure an integer stack any more since it contains integers and operators. Let's called it the output stack.

Indeed if you finish the above trace (with the given modifications), you'll get

```
[0, 1, -, 2, 3, *, +, 4, 5, /, 6, *, -]
```

on your sheet of paper.

Another thing to note is that you only add stuff to the output stack during the process to convert infix to postfix/RPN expression. For the case of evaluating the infix expression, you have to pop integers off the integer stack. So that's another thing different.

But wait ... once the conversion is done, you probably want to read the resulting postfix/RPN expression. Hang on there ...

You always read an expression left-to-right. So if the output stack looks like this:

```
[0, 1, -, 2, 3, *, +, 4, 5, /, 6, *, -]
```

how are you going start reading `0` since it's at the bottom of the stack??? I need to start reading from the bottom of stack.

Hmmm ... I put stuff into one end of the output stack. Then I take things out the other end. AHA! ...

I need an output *queue*, and not an output stack.

I'll leave it to you to write down the algorithm since it's very similar to the algorithm to evaluate an infix expression.

Exercise 106.21.1. Design an algorithm that converts an infix expression to prefix/polish notation. □

Exercise 106.21.2. Design an algorithm that converts an postfix/RPN expression to an infix expression. □

106.22 Stack machine debug: stack-machine.tex

In the previous section, you notice that there's a common method of computation where you operator on the top two values of a stack and then put the result back to the stack. This type of computation appears frequently in computer science. Because of that, if such a computation is needed in what you're tryng to computation, you might want to add this feature to your stack, so that you stack becomes what is known as a **stack machine**.

stack machine

A **stack machine** is a stack that has the usual `stack.push(v)` and `v = stack.pop()` operation, but it also has `stack.add()`, `stack.subtract()`, `stack.mult()`, `stack.div()`, `stack.mod()`, etc. In the case of an integer stack machine, those are some of the common operation. But of course you can add more to your stack machine if you like. For instance you can add the exponentiation operation or a bit shift operation.

stack machine

The above operation (`add()`, etc.) will apply the relevant operator to the top two values on the stack (these two values are removed from the stack) and the result is pushed back onto the stack.

Such a computational model is implemented in software or in hardware. In software, such a computation model is present in compilers and virtual machines.

If you prefer, instead of the above, you can have `stack.op('+')`, `stack.op('-')`, `stack.op('*')`, `stack.op('/')`, `stack.op('%')`, etc.

Exercise 106.22.1. Write a `IntStackMachine` class. Use it to compute `1 + 2 - 3 * 4 + 5` by doing this:

```
IntStackMachine s;
s.push(1);
std::cout << s << std::endl; // prints [1]
s.push(2);
std::cout << s << std::endl; // prints [1, 2]
s.add();
std::cout << s << std::endl; // prints [3]
s.push(3);
std::cout << s << std::endl; // prints [3, 3]
s.push(4);
std::cout << s << std::endl; // prints [3, 3, 4]
s.mult();
std::cout << s << std::endl; // prints [3, 12]
s.subtract();
std::cout << s << std::endl; // prints [-9]
s.push(5);
std::cout << s << std::endl; // prints [-9, 5]
```

```
s.add();
std::cout << s << std::endl; // prints [-4]
```

□

Note that a stack machine evaluates one operator at a time – it's not that smart. The responsibility to take an expression, break up into steps and feed the operations one at a time to stack machine lies somewhere else.

Exercise 106.22.2. Add exponentiation to your stack machine: `stack.pow()`. □

Exercise 106.22.3. Using `0` for false and non–zero for true, add `stack.ifelse()` to your stack machine. This is a ternary operator. The top of the stack is an integer value that represents true/false and is followed by two integer values `a` and `b`. If the top of stack is true, the the computation pushes `a` onto the stack; otherwise `b` is pushed onto the stack. □

Exercise 106.22.4. Clean up your infix evaluation using a stack machine. □

106.23 Long integer debug: long-integer.tex

Recall (see CISS245) that you can model a long integer (some people call it a big integer) with an array (or vector) of integers where each integer in the array is a digit in the integer being modeled. (Of course instead of storing a single digit in one cell of the array/vector of integers, you can store for instance 9 digits.) Is this the best container to use?

Look at the addition algorithm for integers. You can really traversing the digits of two long integers starting from the lowest order digit to the highest. The resulting long integer is also filled in the first direction. This is the same for addition.

What about multiplication? If you do 2 or 3 multicolumn multiplication of integers, you'll see that you also traverse the digits from the lowest order digit to the highest.

Division? The standard method for division is long division. If you do 2 or 3 long divisions, you'll see that you actually have to look at the highest order digit first.

Aha!

This means that the right data structure to use is the doubly linked list. Note that in this case, the digits are stored in the linked list. There's no key.

(In serious number crunching, `std::vector` *is* used because for those applications – which includes breaking codes – speed is essesntial. For that kind of work, after a good algorithm is found, say a $O(n)$ and a $O(n^2)$ algorithm are found and the $O(n)$ is chosen, they still care about whether the algorithm is roughly $10n$ or $12n$.)

Exercise 106.23.1. Implement a `LongInt` class using doubly linked list. □

Exercise 106.23.2. Implement a `Polynomial` class using doubly linked list. □

Exercise 106.23.3. A matrix is a rectangular array of values. Viewing each row as a singly linked list, implement a matrix class. Note again that matrices are used a lot in number crunching cryptographic and scientific applications. In those cases speed might be more important than saving memory. However some matrices in scientific applications are so huge that they have to be broken up not because of lack of memory (or memory fragmentation) but because the system does not support such a huge chunk of contiguous memory. Another reason for breaking it up is to allow better parallel computation. However mstrices are usual broken up into blocks and not rows. Furthermore matrices are frequently processed column-wise as well as row-wise. □

106.24 Sparse long integer debug: sparse-long-integer.tex

Exercise 106.24.1. What if there are lots of zeroes in your integer? Say the integer is

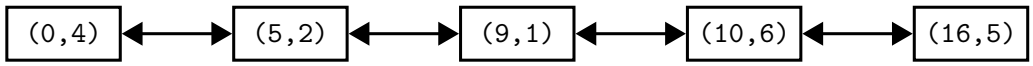
`50000061000200004`

Assume that an `int` 4 bytes (i.e., 32 bits), how many bytes are used when you model the above integer (with 17 digits) an array of `int`s? What if there are n digits?

Now the above integer is the same as

$$5 \cdot 10^{16} + 6 \cdot 10^{10} + 1 \cdot 10^{9} + 2 \cdot 10^{5} + 4 \cdot 10^{0}$$

So we can have (doubly linked) nodes where each nonzero term corresponds to one node, storing the exponent (of 10) and the digit of the term in the node:



□

Assume that an `int` and a pointer uses 4 bytes (i.e., 32 bits), how many bytes are used when you model the above integer (with 17 digits) with this new method? What if there are n digits bit but only rn ($r < 1$) are nonzero?

Therefore if r is the fraction of nonzero terms in a long integer with n digits, then the second method uses less memory if

$$rn \cdot 4 \cdot 4 < n \cdot 4$$

i.e.,

$$r < 1/4$$

In particular if you want to store

$$1 \times 10^{100}$$

The second method uses 16 bytes while the first method uses ... $101 \cdot 4 = 404$ bytes ... ouch. The second method is way better.

Exercise 106.24.2. Implement a polynomial class using the above idea. In the case of polynomials used in cryptography and signal processing (example: error correction codes), you *do* see a lot of cases where most of the coefficients are zero. For instance the binary polynomial

$$x^{571} + x^{10} + x^{5} + x^{2} + 1$$

is recommended for use in elliptic curve crygtography. (See https://www.secg.org/sec2-v2.pdf, table 3.) A binary polynomials is a polynomial where the coefficients are either 0 or 1. □

106.25 Sparse matrix

debug: sparse-matrix.tex

Here's a matrix:

$$\begin{bmatrix} 1 & 3 & -1 & 2 \\ 7 & 0 & 8 & 0 \\ 8 & 0 & 9 & 2 \\ 1 & 1 & 8 & 0 \end{bmatrix}$$

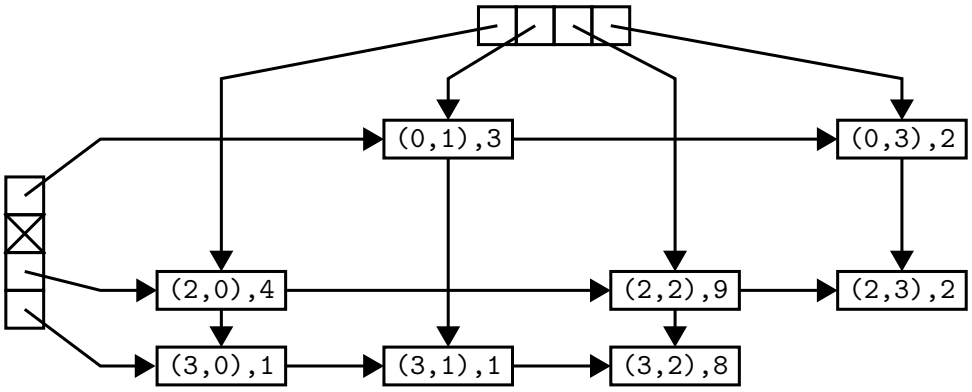
This is a rectangular array of numbers. We can use a 2D array as a container for these values.

Suppose there are lots of zeroes in the matrix like this:

$$\begin{bmatrix} 0 & 3 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 4 & 0 & 9 & 2 \\ 1 & 1 & 8 & 0 \end{bmatrix}$$

Such a matrix is a **sparse matrix**. When the percentage of 0s is extremely high, instead of a 2D array, one can use two arrays of doubly linked list like this:

sparse matrix



Note that each doubly linked node points to two nodes, the next node in the same row and the next node in the same column. Furthermore, each node appears twice, once in the array of linked list for rows and once in the array of linked list for columns.

For instance look at the $(2, 0)$ entry of the matrix. The value there is 4. Therefore for this nonzero entry of the matrix, we have a node with value `(2,0),4`. Each node is in two linked list, a row listed list and a column linked list. A row linked list links up nodes in the same row and a column linked lists up nodes in the same column. Clearly each node has two pointers, one pointing to the node next in the row, and another pointing to the node next in the column.

Get it?

If a matrix is sparse, you can also use a hashtable – I'll talk about this later.

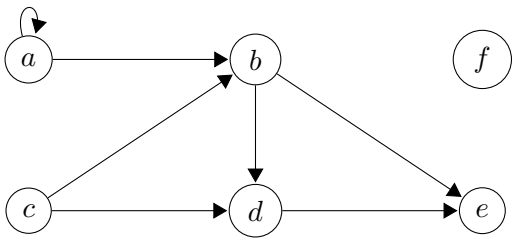
106.26 Graphs: Adjacency matrix and adjacency list

debug: adjacency-list.tex

(This is a quickie on graphs – it's an extremely huge subject. We'll come back to graphs again much later.)

Here's a **directed graph**:

directed graph



It's basically a bunch of dots/circles and lines with arrows. An **undirected graph** is a graph where there are no arrows on the line: in that case you think of the lines as bi-directional. The techical term for the dots/circles is **nodes** or **vertices** and the technical term of the lines is **directed or undirected edges**. Directed edges are also called **arcs**.

undirected graph

~~vertex~~es

directed or undirected edges

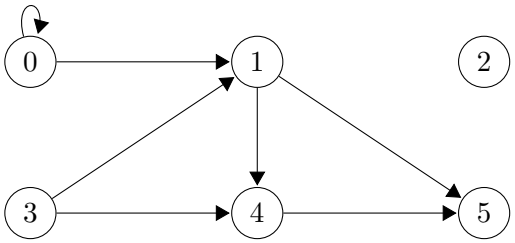
arcs

Graphs are extremely important in computer science. Here's a list of some applications:

(a) Road networks are graphs
(b) Social graph (describing association between people on the internet) is graphs
(c) Game trees are graph – each each node describes a particular state of a game and an edge denotes a move.

Here's an important question on graphs: Given two nodes, what is the shortest path from one to the other (if it exists). In the case where the one of the state is the current state of a game of chess and the target is a state (or more than one state) where you win the chess game, you're basically asking "is there a sequence of moves that will allow me to checkmate my opponent".

There are several says to provide a mathematical model of a graph. The easiest is probably to use a matrix. First let me relabel the nodes like this



Now if I can go from node i to node j, then in the matrix I put a 1 at row i and column j. For instance 0 can go to 0. I'm numbering the rows starting with 0; likewise for the columns. So I get

$$\begin{bmatrix} 1 & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix}$$

0 can also go to 1. So ...

$$\begin{bmatrix} 1 & 1 & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix}$$

0 can't go to the others. So ...

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix}$$

1 can go to 4 and 5:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix}$$

2 can't go anywhere:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix}$$

3 can go to 1 and 4:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ & & & & & \\ & & & & & \end{bmatrix}$$

4 can go to 5:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ & & & & & \end{bmatrix}$$

And finally 5 can't go anywhere:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

TADA! That's call the **adjacency matrix** of the given graph. Of course we can then use a 2D array to model the above matrix. Or you can use an array of pointers to linked lists. See section on matrices.

adjacency matrix

Exercise 106.26.1. (a) Given nodes i and j of a graph what is the runtime to compute if there's an edge from i to j if the graph is represented using an adjacency matrix?

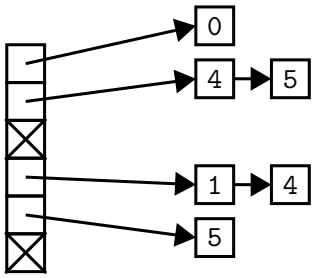
(b) The in-degree is the number of nodes that goes to i. What is the runtime to compute the in-degree is the graph is represented using an adjacency matrix?

(c) The out-degree is the number of nodes that i can go to. What is the runtime to compute the in-degree is the graph is represented using an adjacency matrix?

Note that the above matrix has lots of 0s. In fact there are only 7 1s in the 25 possible slots. Of course the 7 1s corresponds to the 7 edges of the graph. The

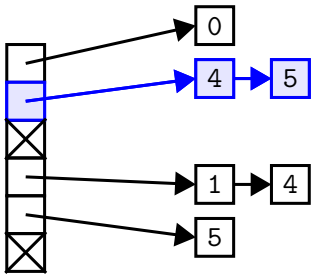
sparsity of 1s is yelling at you do to a better job ...

Another way to represent the graph (especially when the adjacency matrix is sparse) is to use an array of linked lists. A picture tells a thousand words:



This array of linked list is called an **adjacency list**. The data structure here is a list of singly linked lists of numbers repreesenting the nodes of the graph. If the array of linked list is called `x`, then `x[1]` contains 4 and 5:

adjacency list



This tells us that there are two directed edge from 1, one going to 4 another going to 5.

Exercise 106.26.2. Suppose you have a graph with n nodes, each integer and pointer in your computer consumes 4 bytes of memory.

(a) How much memory is needed for an adjacency matrix? [ANSWER: $n^2 \cdot 4$.]
(b) How much memory is needed for the adjacency linked list container assuming that each node is pointing to at most k nodes? [ANSWER: $n \cdot 4 + n \cdot k \cdot 8$.]
(c) With the information above, what condition do you have on k such that the amount of memory used for an adjacency linked list container is less than the memory used by an adjacency matrix?
(d) What is the runtime to check if there's an edge going from i to j if you use an adjacency matrix?
(e) What is the runtime to check if there's an edge going from i to j if you use an adjacency list?

□

Exercise 106.26.3. (a) Given nodes i and j of a graph what is the runtime to compute if there's an edge from i to j if the graph is represented using an adjacency list? Assume that there are n nodes and all lists have lengths at most k.

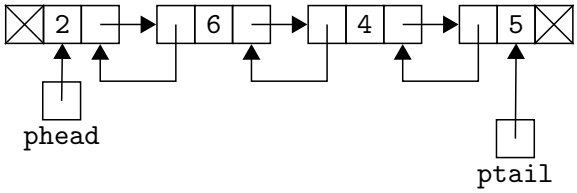
(b) The in-degree is the number of nodes that goes to i. What is the runtime to compute the in-degree is the graph is represented using an adjacency list? (See assumption above).

(c) The out-degree is the number of nodes that i can go to. What is the runtime to compute the in-degree is the graph is represented using an adjacency list? (See assumption above).

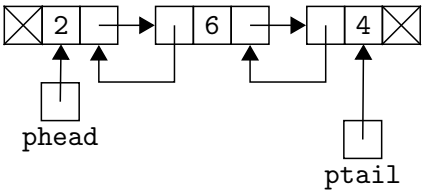
106.27 More exercises debug: exercises.tex

Exercise 106.27.1. I'm going to make some changes to the doubly linked list. Here's a doubly linked list (without sentinel nodes):

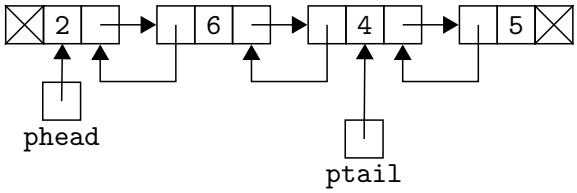
debug: exercises/exercises0/question.tex



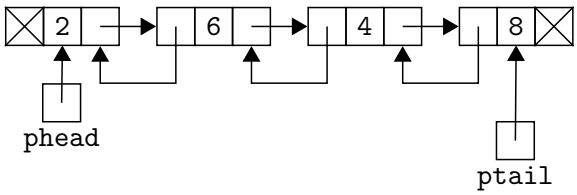
The usual delete tail will result in this:



Write a doubly linked list class that does this instead:

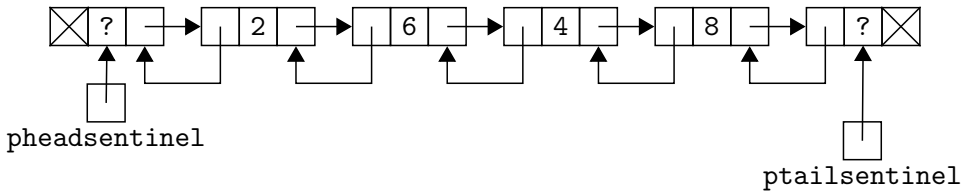


When I insert 8 as tail, I don't have to allocate memory. I just use the node with 5, but I replace the 5 with 8:

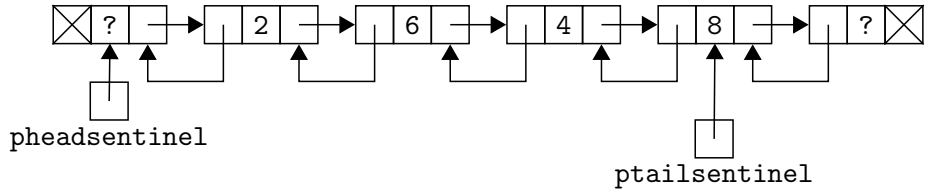


In other words, we delay the deallocation of head and tail nodes. We can think of the used but not deallocated nodes as extra nodes waiting to be reused. When we do want to release the memory, we call the `shrink_to_fit()`. Likewise, the same tbing happens at the end.

Next, add sentinel nodes:

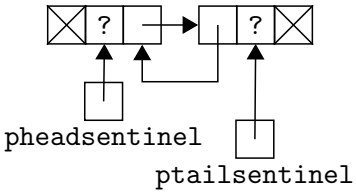


Note that I'm renaming `phead` to `pheadsentinel` and `ptail` to `ptailsentinel` (The `?` denotes integer values we don't care about.) And if I delete the tail, the above becomes

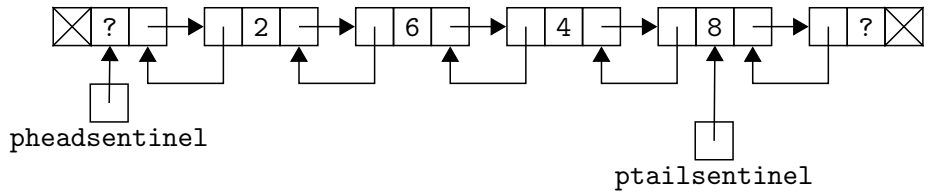


At this point, the size of the linked list is (of course) 3 and the capacity is 4.

Of course the linked list starts off as an empty list that looks like this (there are two sentinel nodes)



Objects of this type of linked list have `size()` and `capacity()` that returns the obvious integer values. For instance for this case:



the size is 3 and the capacity is 4.

Implement this type of linked list. (Go to solution, page 5101) □

Exercise 106.27.2. Write this function

debug: exercises/exercises1/question.tex

```
template < typename T >
```

```
void mergesort(std::list< T > & x);
```

that performs mergesort on linked lists. (Go to solution, page 5102) □

Exercise 106.27.3. Write a sorted doubly linked list class. (Go to solution, page 5103) □

debug: exercises/exercises2/question.tex

Exercise 106.27.4. skip list (Go to solution, page 5104) □

debug: exercises/exercises3/question.tex

Exercise 106.27.5. Random maze generation

debug: exercises/exercises4/question.tex

The goal is to generate a maze in a 2D grid. Suppose you have the following 4x4 grid:

```
+-+-+-+-+
| | | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
```

You can think of the above as a grid of 4-by-4 rooms. Initially think of each room as completely sealed with walls. You can think of building a maze as punching a hole in the wall. For instance here's a hole, from (0,0) to (0,1):

```
+-+-+-+-+
|  x| | |
+-+-+-+-+
| | | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
```

I'm using `x` to match where I am in the grid. And now I punch another hole from (0,1) to (1,1):

```
+-+-+-+-+
|   | | |
+-+ +-+-+
| |x| | |
+-+-+-+-+
| | | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
```

Now if I punch (1,1) to (1,2) to (1,3) to (0,3) to (0,2) I get this:

```
+-+-+-+-+
|   |x  |
+-+ +-+ +
| |     |
+-+-+-+-+
| | | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
```

At this point, I'm stuck – see the `x` in the incomplete maze. I'm stuck in the sense that I can only punch two walls but each will go into a room that is already visited. Again see the diagram above. What I need to do is to go backwards. Going one step back, I get to this point:

```
+-+-+-+-+
|   |  x|
+-+ +-+ +
| |     |
+-+-+-+-+
| | | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
```

But I'm still stuck: I don't any walls to punch (you have to stay in the grid so you cannot punch the perimeter wall). Butif I cgo one more step back I arrive at this point:

```
+-+-+-+-+
|   |   |
+-+ +-+ +
| |    x|
+-+-+-+-+
| | | | |
+-+-+-+-+
| | | | |
+-+-+-+-+
```

At this point I can go from (1,3) to (2,3):

```
+-+-+-+-+
|   |   |
+-+ +-+ +
| |     |
+-+-+-+ +
| | | |x|
+-+-+-+-+
| | | | |
+-+-+-+-+
```

To be able to build the maze I will need the following idea. Make sure you study and think about it and make changes/corrections/modification when necessary.

```
ALGORITHM: build_maze
INPUT: (r,c) - the starting point

let UNVISITED be the container of unvisited cells:
    initially this should be all cells except for (r,c)
let PATH be the container containing only (r,c)
let VISITED be a container containing only (r,c)
let PUNCHED be an empty container of "(r0,c0) -- (r1,c1)"
    that represents punched walls

while PATH is not empty:
    look at the last step stored in PATH -- call it x
    look at all the surrounding cells around x
    the available cells to go to are the
        cells to the north, south, east, west of x
        which are within the grid and not visited yet,
        i.e., in UNVISITED.
    if there is at least one available cell:
        randomly choose one available cell -- call it y
        store y in VISITED and add that to PATH
        remove y from UNVISITED
        store x--y (the punched wall) in PUNCHED
    else:
        there are are no available cells
        we have to go backwards -- remove x from PATH
```

Note that the `PATH` is a stack (this should be clear from the way I describe how I use `PATH`). (Do I need both `VISITED`. and `UNVISITED`?)

Note that you can view the maze as a graph. The nodes are of the form (r, c). An edge joins `(r0,c0)` to `(r1,c1)` is the same as you can go from room `(r0,c0)` to `(r1,c1)`. The maze can be described by a vector of `PunchedWall`. Call the function

```
class Cell
{
public:
    int r, c;
};

class PunchedWall
{
public:
    Cell c0, c1;
};

// Return an adjacency list describing the maze
// The maze is n-by-n and (r,c) is the starting point of the
// maze.
std::vector< PunchedWall > build_maze(int n,
                                      int r, int c);

void print_maze( int n, const std::vector<PunchedWall> & v)
```

The `print_maze` will print a maze in the following manner:

```
+-+-+-+-+
|   |   |
+-+ +-+ +
| |     |
+ +-+-+ +
|     | |
+-+ +-+ +
|       |
+-+-+-+-+
```

(Go to solution, page 5105) □

Here's the (obvious) main():

```
int main()
{
    int n, r, c;
    std::cin >> n >> r >> c;

    punched_walls = build_maze(int n, int r, int c);
    print_maze(n, punched_walls);

    return 0;
}
```

Index

abstract data type, 5106
adjacency list, 5217
adjacency matrix, 5216
arcs, 5214

cycle, 5132

data structure, 5106
directed graph, 5214
directed or undirected edges, 5214

FIFO, 5155
first-in-first-out, 5155
Floyd's tortoise and hare algorithm, 5132

head, 5108
head sentinel, 5137

infix notation, 5178

last-in-first-out, 5106
left associative, 5187
left-to-right associative, 5187
LIFO, 5106

nodes, 5214

Polish notation, 5178
popping, 5153
postfix notation, 5183
prefix notation, 5178

reverse Polish notation, 5183

self-organizing container, 5106
sentinel node, 5137
singly linked list, 5108
singly linked node, 5108
sparse matrix, 5212
stack machine, 5207

tail, 5108
tail sentinel, 5138
top, 5106, 5153

undirected graph, 5214

vertices, 5214