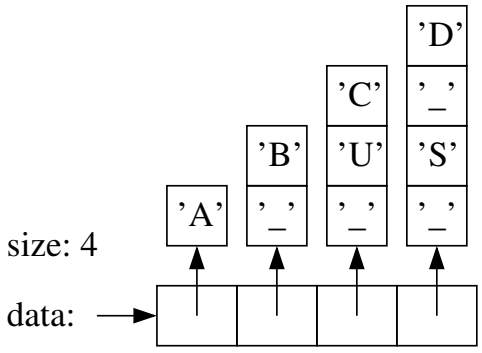
CSCI-1200 Data Structures — Fall 2016
Test 2 — Solutions

1 Dynamically Allocated & Templated Stairs [/ 28]

In this problem you will write a simple class to build a staircase-shaped storage shelf. Here's an example usage of the class, which constructs the diagram on the right.

```
Stairs<char> s(4,'_');
s.set(0,0,'A');
s.set(1,1,'B');
s.set(2,2,'C');
s.set(3,3,'D');
s.set(2,1,'U');
s.set(3,1,'S');
```



1.1 `Stairs` Class Declaration [/ 14]

First, fill in the blanks in the class declaration:

Solution:

```
template <class T> class Stairs {

  public:
    // constructor

    Stairs(int s, const T& val);

    // destructor

    ~Stairs();

    // prototypes of 2 other important functions related to the constructor & destructor

    Stairs(const Stairs& s);
    Stairs& operator=(const Stairs& s);

    // modifier
    void set(int i, int j, const T& val) { data[i][j] = val; }

    /* NOTE: other Stair functions omitted */
  private:
    // representation

    int size;
    T** data;

};
```

1.2 `Stairs` Constructor [/ 9]

Now write the constructor, as it would appear outside of the class declaration (because the implementation is > 1 line of code).

Solution:

```
template <class T> Stairs<T>::Stairs(int s, const T& val) {
  size = s;
  data = new T*[s];
  for (int i = 0; i < s; i++) {
    data[i] = new T[i+1];
    for (int j = 0; j <= i; j++) {
      data[i][j] = val;
    }
  }
}
```

1.3 `Stairs` Destructor [/ 5]

Now write the destructor, as it would appear outside of the class declaration (because the implementation is > 1 line of code).

Solution:

```
template <class T> Stairs<T>::~Stairs() {
  for (int i = 0; i < size; i++) {
    delete [] data[i];
  }
  delete [] data;
}
```

2 Comparing Linked List Pointers w/ Recursion [/ 32]

Ben Bitdiddle is working on a software project for essay writing using a doubly-linked chain of nodes. His initial `Node` class is on the right.

```
class Node {
public:
  std::string word;
  Node* next;
  Node* prev;
};
```

One of the features of his software allows a user to compare the location of two words within the document and say which word appears first. Ben plans to implement this using two helper functions: `search` and `compare`.

2.1 Searching for a Word [/ 7]

First, let's write the `search` function, which takes in two arguments: a pointer to the first `Node` in the document (word chain) and the specific `word` we're looking to find. The function returns a pointer to the first `Node` containing that `word`. Use *recursion* to implement this function.

Solution:

```
Node* search(Node* head, const std::string& word) {
  if (head == NULL) return NULL;
  if (head->word == word) {
    return head;
  }
  return search (head->next,word);
}
```

If the `Node` chain contains n elements, what is the running time of the `search` function?

Solution: O(n)

2.2 Comparing Positions within the Node Chain [/ 8]

Next, let's implement the `compare` function. This function takes in two `Node` pointers and returns true if the first argument appears closer to the front of the list than the second argument. For example, let's say a chain of word `Node`s named `sentence` contains:

```
the quick brown fox jumps over the lazy dog
```

Here's an example using the `search` and `compare` functions:

```
Node* over  = search(sentence,"over");
Node* quick = search(sentence,"quick");
Node* lazy  = search(sentence,"lazy");

assert (compare(quick,over) == true);
assert (compare(over,quick) == false);
assert (compare(quick,lazy) == true);
assert (compare(lazy,over)  == false);
```

Again using *recursion*, implement the `compare` function.

Solution:

```
bool compare(Node *a, Node *b) {
  if (a == NULL) return false;
  if (b == NULL) return false;
  if (a == b) return false;
  if (a->next == b) return true;
  return compare(a->next,b);
}
```

If the `Node` chain contains n elements, what is the running time of the `compare` function?

Solution: O(n)

Improving Word Position Comparison Performance

Alyssa P. Hacker stops by to help, and suggests that Ben switch to a different data structure if he is frequently comparing word positions within a long essay.

But Ben's a stubborn guy. Instead of switching to a different data structure, he has a plan to augment his list structure to improve the running time of `compare`. Ben explains that the new `distance` member variable in each node will indicate how far away the node is from the front of the list.

```
class Node {
public:
  std::string word;
  Node* next;
  Node* prev;
  float distance;
};
```

Here's Ben's new compare function:

```
bool compare_fast(Node *a, Node *b) {
  return a->distance < b->distance;
}
```

Ben reassures Alyssa that he'll add some error checking to this code.
SIDE NOTE: Hopefully your implementation of the original compare function has some error checking!

But Alyssa is more concerned about how this addition to the data structure will impact performance when the essay or sentence is edited. She says he can't afford to change the `distance` in all or many `Node`s in the data structure any time a small edit is made to the document.

Ben explains that the `push_back` function will assign the distance of the new `Node` to be the distance of the last `Node` in the chain plus 10.0. And similarly, `push_front` will assign the new `Node` to be the distance from the first `Node` minus 10.0. *BTW, negative distance values are ok.* Finally, Ben says the `insert_between` function (on the next page) can similarly be implemented without editing the distance value in any existing `Node`!

2.3 Implementing `insert_between` and Maintaining Fast Comparisons [/ 17]

Continuing with the previous example, here's a quick demonstration of how this function works:

```
bool success = insert_between(sentence,"the","lazy","VERY");
assert (success);
Node* VERY = search(sentence,"VERY");
assert (compare(VERY,lazy)       == true);
assert (compare(quick,VERY)      == true);
assert (compare_fast(VERY,lazy)  == true);
assert (compare_fast(quick,VERY) == true);
success = insert_between(sentence,"quick","fox","RED");
assert (!success);
```

And here's the contents of the `sentence` variable after the above fragment of code:

```
the quick brown fox jumps over the VERY lazy dog
```

Implement `insert_between`. And yes, use *recursion*.

Solution:

```
bool insert_between(Node *head, const std::string& before, const std::string& after, const std::string& word) {
  if (head == NULL) return false;
  if (head->next == NULL) return false;
  if (head->word == before && head->next->word == after) {
    Node* tmp = new Node;
    tmp->word = word;
    tmp->prev = head;
    tmp->next = head->next;
    tmp->next->prev = tmp;
    tmp->prev->next = tmp;
    tmp->distance = (tmp->next->distance + tmp->prev->distance) / 2.0;
    return true;
  } else {
    return insert_between(head->next,before,after,word);
  }
}
```

3 Erase Middles [/ 20]

Write a function named `erase_middles` that takes in 2 arguments: an STL list named `data` and a `value`. The function should remove all instances of `value` from `data`, except the first and the last instances. The function returns the number of removed elements. For example, if `data` initially contains:

```
5 2 5 2 3 4 3 2 5 2 3 2 3 4 2 5
```

A call to `erase_middles(data,5)` will return 2 and now `data` contains:

```
5 2 2 3 4 3 2 2 3 2 3 4 2 5
```

And then a call to `erase_middles(data,2)` will return 4 and `data` contains:

```
5 2 3 4 3 3 3 4 2 5
```

Solution:

```
template <class T>
int erase_middles(std::list<T>& data, const T& val) {
  bool found_first = false;
  typename std::list<T>::iterator prev = data.end();
  typename std::list<T>::iterator itr = data.begin();
  int count = 0;
  while (itr != data.end()) {
    if (*itr == val) {
      if (prev != data.end()) {
        data.erase(prev);
        count++;
      }
      if (found_first == false) {
        found_first = true;
      } else {
        prev = itr;
      }
    }
    itr++;
  }
  return count;
}
```

4 Debugging Skillz [/ 17]

For each program bug description below, write the letter of the most appropriate debugging skill to use to solve the problem. Each letter should be used at most once.

A) get a backtrace
B) add a breakpoint
C) use step or next
D) add a watchpoint
E) examine different frames of the stack
F) reboot your computer
G) use Dr Memory or Valgrind to locate the leak
H) examine variable values in gdb or lldb

Solution: E A complex recursive function seems to be entering an infinite loop, despite what I think are perfect base cases.

Solution: G The program always gets the right answer, but when I test it with a complex input dataset that takes a long time to process, my whole computer slows down.

Solution: A I'm unsure where the program is crashing.

Solution: H I've got some tricky math formulas and I suspect I've got an order-of-operations error or a divide-by-zero error.

Solution: D I'm implementing software for a bank, and the value of a customer's bank account is changing in the middle of the month. Interest is only supposed to be added at the end of the month.

Select one of the letters you did not use above, and write a concise and well-written 3-4 sentence description of a specific situation where this debugging skill would be useful.

Solution: B) Once you've found the general area of the problem, it can be helpful to add a breakpoint shortly before the crash, so you can examine the situation more closely. C) Once you've decided the state of the program is reasonable, you can advance the program one line at a time using `next` or `step` into a helper function that may be causing problems. Rebooting your computer is unlikely to fix a bug in your own code.