Representing a Sequence of Data

- Sequence – an ordered collection of items (position matters)
  - we will look at several types: lists, stacks, and queues
- Most common representation = an array
- Advantages of using an array:
  - easy and efficient access to any item in the sequence
    - \texttt{item[i]} gives you the item at position \texttt{i}
    - every item can be accessed in constant time
    - this feature of arrays is known as random access
  - very compact (but can waste space if positions are empty)
- Disadvantages of using an array:
  - have to specify an initial array size and resize it as needed
  - difficult to insert/delete items at arbitrary positions
    - ex: insert 63 between 52 and 72

\[
\text{item} \quad 31 \quad 52 \quad 72 \quad \ldots
\]
**Alternative Representation: A Linked List**

- Example:

```
items
```

```
31 ———> 52 ———> 72 ———> null
```

- A linked list stores a sequence of items in separate *nodes*.

- Each node contains:
  - a single item
  - a “link” (i.e., a reference) to the node containing the next item

```
example node:
```

```
31
```

- The last node in the linked list has a link value of *null*.

- The linked list as a whole is represented by a variable that holds a reference to the first node (e.g., `items` in the example above).

---

**Arrays vs. Linked Lists in Memory**

- In an array, the elements occupy consecutive memory locations:

```
item
```

```
31 52 72 ...
```

```
0x100 0x104 0x108
```

- In a linked list, each node is a distinct object on the heap. The nodes do *not* have to be next to each other in memory. That's why we need the links to get from one node to the next.

```
items
```

```
31 ———> 52 ———> 72 ———> null
```

```
items
```

```
0x520 0x812 0x208
```

```
0x520 0x812 0x208
```

```
31 52 72
```

```
0x520 0x812 0x208
```

```
72 null
```

```
0x520
```

```
0x812
```

```
0x208
```

```
null
```
Linked Lists in Memory

Here’s how the above linked list might actually look in memory:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x200</td>
<td>0x520</td>
</tr>
<tr>
<td>0x204</td>
<td></td>
</tr>
<tr>
<td>0x208</td>
<td>72</td>
</tr>
<tr>
<td>0x212</td>
<td>null</td>
</tr>
<tr>
<td>0x216</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x520</td>
<td>31</td>
</tr>
<tr>
<td>0x524</td>
<td>0x812</td>
</tr>
<tr>
<td>0x528</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0x812</td>
<td>52</td>
</tr>
<tr>
<td>0x816</td>
<td>0x208</td>
</tr>
</tbody>
</table>

Features of Linked Lists

- They can grow without limit (provided there is enough memory).
- Easy to insert/delete an item – no need to “shift over” other items.
  - for example, to insert 63 between 52 and 72, we just modify the links as needed to accommodate the new node:

  before:

  ![Diagram of linked list before insertion]

  after:

  ![Diagram of linked list after insertion]

- Disadvantages:
  - they don’t provide random access
  - need to “walk down” the list to access an item
  - the links take up additional memory
A String as a Linked List of Characters

- Each node in the linked list represents one character.
- Java class for this type of node:
  ```java
  public class StringNode {
      private char ch;
      private StringNode next;
      ...
  }
  ```
- The string as a whole will be represented by a variable that holds a reference to the node containing the first character.
  ```java
  StringNode str1;   // shown in the diagram above
  ```
- Alternative approach: use another class for the string as a whole.
  ```java
  public class LLString {
      StringNode first;   // (we will not do this for strings)
      ...
  }
  ```

A String as a Linked List (cont.)

- An empty string will be represented by a null value.
  ```java
  StringNode str2 = null;
  ```
- We will use static methods that take the string as a parameter.
  - e.g., we will write `length(str1)` instead of `str1.length()`
  - outside the class, need the class name: `StringNode.length(str1)`
- This approach is necessary so that the methods can handle empty strings.
  - if `str1 == null`, `length(str1)` will work,
    but `str1.length()` will throw a `NullPointerException`
- Constructor for our `StringNode` class:
  ```java
  public StringNode(char c, StringNode n) {
      ch = c;
      next = n;
  }
  ```
  (see `-cscie119/examples/sequences/StringNode.java`)
A Linked List Is a Recursive Data Structure

- Recursive definition of a linked list: a linked list is either
  a) empty or
  b) a single node, followed by a linked list

- Viewing linked lists in this way allows us to write recursive
  methods that operate on linked lists.

- Example: length of a string
  length of "cat" = 1 + the length of "at"
  length of "at" = 1 + the length of "t"
  length of "t" = 1 + the length of the empty string (which = 0)

- In Java:
  ```java
  public static int length(StringNode str) {
      if (str == null)
          return 0;
      else
          return 1 + length(str.next);
  }
  ```

Tracing `length()`

```java
public static int length(StringNode str) {
    if (str == null)
        return 0;
    else
        return 1 + length(str.next);
}
```

- Example: StringNode.length(str1)
Getting the Node at Position i in a Linked List

• `getNode(str, i)` — a private helper method that returns a reference to the ith node in the linked list (i == 0 for the first node)

• Recursive approach:
  - node at position 2 in the linked list representing “linked”
    = node at position 1 in the linked list representing “inked”
    = node at position 0 in the linked list representing “nked”
    (return a reference to the node containing ’n’)

• We’ll write the method together:
  ```java
  private static StringNode getNode(StringNode str, int i) {
  ```

Review of Variables

• A variable or variable expression represents both:
  • a “box” or location in memory (the address of the variable)
  • the contents of that “box” (the value of the variable)

• Practice:

<table>
<thead>
<tr>
<th>expression</th>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>0x200</td>
<td>0x520 (reference to the ’d’ node)</td>
</tr>
<tr>
<td>str.ch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>str.next</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More Complicated Expressions

- Example: `temp.next.ch`
  - Start with the start of the expression: `temp.next`
    - It represents the `next` field of the node to which `temp` refers.
    - Address =
    - Value =
  - Next, consider `temp.next.ch`
    - It represents the `ch` field of the node to which `temp.next` refers.
    - Address =
    - Value =

Dereferencing a Reference

- Each dot causes us to dereference the reference represented by the expression preceding the dot.
- Consider again `temp.next.ch`
  - Start with `temp`: `temp.next.ch`
  - Dereference: `temp.next.ch`

```
Dereferencing a Reference (cont.)

- Get the `next` field: `temp.next.ch`

```
\text{str} 0x200 \rightarrow \quad \text{temp} 0x204
\hspace{10pt} \quad \text{temp} 0x204
```

- Dereference: `temp.next.ch`

```
\text{str} 0x200 \rightarrow \quad \text{temp} 0x204
\hspace{10pt} \quad \text{temp} 0x204
```

- Get the `ch` field: `temp.next.ch`

```
\text{str} 0x200 \rightarrow \quad \text{temp} 0x204
\hspace{10pt} \quad \text{temp} 0x204
```

More Complicated Expressions (cont.)

- Here’s another example: `str.next.next`
  - address = ?
  - value = ?
Assignments Involving References

• An assignment of the form
  \[ \text{var1} = \text{var2}; \]
  takes the value of \text{var2} and copies it into the location in memory given by the address of \text{var1}.

• Practice:
  1) \text{str.next} = \text{temp.next};
  2) \text{temp} = \text{temp.next};

• What happens if we do the following?
  1) \text{str.next} = \text{temp.next};
  2) \text{temp} = \text{temp.next};

Assignments Involving References (cont.)

• Beginning with the original diagram, if \text{temp} didn’t already refer to the ‘o’ node, what assignment would we need to perform to make it refer to that node?

  1) \text{str.next} = \text{temp.next};
  2) \text{temp} = \text{temp.next};
Creating a Copy of a Linked List

- `copy(str)` – create a copy of `str` and return a reference to it

- Recursive approach:
  - base case: if `str` is empty, return `null`
  - else: copy the first character
    make a recursive call to copy the rest

```java
public static StringNode copy(StringNode str) {
    if (str == null) { // base case
        return null;
    }
    // create the first node of the copy, copying the first character into it
    StringNode copyFirst = new StringNode(str.ch, null);
    // make a recursive call to get a copy the rest and store the result in the first node's next field
    copyFirst.next = copy(str.next);
    return copyFirst;
}
```

Tracing `copy()`: part I

- Example: `StringNode s2 = StringNode.copy(s1);`

- The stack grows as a series of recursive calls are made:
Tracing `copy()` : part II

- The base case is reached, so the final recursive call returns `null`.
- This return value is stored in the `next` field of the `'g'` node:

```
copyFirst.next = copy(str.next)
```

Tracing `copy()` : part III

- The recursive call that created the `'g'` node now completes, returning a reference to the `'g'` node.
- This return value is stored in the `next` field of the `'o'` node:
Tracing `copy()` : part IV

- The recursive call that created the 'o' node now completes, returning a reference to the 'o' node.
- This return value is stored in the `next` field of the 'd' node:

```
s1: copyFirst str
    `d'
    null

s2: copyFirst str
    `d'

s2: `o'

s2: `g'
  null
```

Tracing `copy()` : part V

- The original call (which created the 'd' node) now completes, returning a reference to the 'd' node.
- This return value is stored in `s2`:

```
s1: copyFirst str
    `d'
    null

s2: copyFirst str
    `d'

s2: `d'

s2: `o'

s2: `g'
  null
```
Tracing `copy()`: Final Result

- `StringNode s2 = StringNode.copy(s1);`
- `s2` now holds a reference to a linked list that is a copy of the linked list to which `s1` holds a reference.

Using Iteration to Traverse a Linked List

- Many tasks require us to traverse or “walk down” a linked list.
- We’ve already seen methods that use recursion to do this.
- It can also be done using iteration (for loops, while loops, etc.).
- We make use of a variable (call it `trav`) that keeps track of where we are in the linked list.

- Template for traversing an entire linked list:
  ```java
  StringNode trav = str; // start with the first node
  while (trav != null) {
      // usually do something here
      trav = trav.next; // move trav down one node
  }
  ```
Example of Iterative Traversal

- `toUpperCase(str)`: converting `str` to all upper-case letters

```
public static void toUpperCase(StringNode str) {
    StringNode trav = str;
    while (trav != null) {
        trav.ch = Character.toUpperCase(trav.ch);
        trav = trav.next;
    }
}
```

(makes use of the `toUpperCase()` method from Java's built-in `Character` class)

Tracing `toUpperCase()`: Part I

Calling `StringNode.toUpperCase(str)` adds a stack frame to the stack:
Tracing `toUpperCase()`: Part II

From the previous page:

```java
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

Results of the first pass through the loop:

```
str
str
str
'tf'
'i'
'n'
'e'
null
```

Tracing `toUpperCase()`: Part III

```
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

Results of the second pass through the loop:

```
str
str
str
'F'
'I'
'n'
'e'
null
```

Results of the third pass:

```
str
str
str
'F'
'I'
'N'
'e'
null
```
Tracing `toUpperCase()`: Part IV

```java
while (trav != null) {
    trav.ch = Character.toUpperCase(trav.ch);
    trav = trav.next;
}
```

Results of the fourth pass through the loop:

```
trav null
str ________
str 'F' ________
str 'I' ________
str 'N' ________
str 'E' null
```

And now trav == null, so we break out of the loop and return:

```
str ________
str 'F' ________
str 'I' ________
str 'N' ________
str 'E' null
```

Deleting the Item at Position i

- Two cases:
  1) `i == 0`: delete the first node by doing
     ```java
     str = str.next;
     ```

```
str ________
str 'j' ________
str 'a' ________
str 'v' ________
```

2) `i > 0`: first obtain a reference to the previous node
   (example for `i == 1`)

```
str ________
str 'j' ________
str 'a' ________
str 'v' ________
prevNode ________
```

What line of code will perform the deletion?
Inserting an Item at Position i

- **Case 1:** \( i == 0 \) (insertion at the front of the list):
  - What line of code will *create* the new node?

\[
\text{newNode} \rightarrow 'f'
\]

\[
\text{str} \rightarrow 'a' \rightarrow 'c' \rightarrow 'e' \rightarrow \text{null}
\]

- What line of code will *insert* the new node?

\[
\text{newNode} \rightarrow 'f'
\]

\[
\text{str} \rightarrow 'a' \rightarrow 'c' \rightarrow 'e' \rightarrow \text{null}
\]

Inserting an Item at Position i (cont.)

- **Case 2:** \( i > 0 \): insert *before* the character currently in posn \( i \)
  - First obtain a reference to the node at position \( i - 1 \):
    (example for \( i == 2 \))

\[
\text{str} \rightarrow 'a' \rightarrow 'c' \rightarrow 'e' \rightarrow \text{null}
\]

\[
\text{prevNode} \rightarrow \text{null}
\]

- What lines of code will insert the character ‘m’?

\[
\text{newNode} \rightarrow 'm'
\]

\[
\text{str} \rightarrow 'a' \rightarrow 'c' \rightarrow 'e' \rightarrow \text{null}
\]

\[
\text{prevNode} \rightarrow \text{null}
\]
Returning a Reference to the First Node

• Both `deleteChar()` and `insertChar()` return a reference to the first node in the linked list. For example:

```java
private static StringNode deleteChar(StringNode str, int i) {
    if (i == 0) // case 1
        str = str.next;
    else { // case 2
        StringNode prevNode = getNode(str, i-1);
        if (prevNode != null && prevNode.next != null)
            prevNode.next = prevNode.next.next;
    }
    return str;
}
```

• They do so because the first node may change.

• Invoke as follows:
  ```java
  str = StringNode.deleteChar(str, i);
  str = StringNode.insertChar(str, i, ch);
  ```

• If the first node changes, `str` will point to the new first node.

Using a “Trailing Reference” During Traversal

• When traversing a linked list, using a single `trav` reference isn’t always good enough.

• Ex: insert `ch = ‘n’` at the right place in this `sorted` linked list:

```
<table>
<thead>
<tr>
<th>char</th>
<th>trav</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>'a'</td>
</tr>
<tr>
<td>'c'</td>
<td>'p'</td>
</tr>
<tr>
<td>null</td>
<td>'z'</td>
</tr>
</tbody>
</table>
```

• Traverse the list to find the right position:

```java
StringNode trav = str;
while (trav != null && trav.ch < ch)
    trav = trav.next;
```

• When we exit the loop, where will `trav` point? Can we insert ‘n’?

• The following changed version doesn’t work either. Why not?

```java
StringNode trav = str;
while (trav != null && trav.next.ch < ch)
    trav = trav.next;
```
Using a “Trailing Reference” (cont.)

• To get around the problem seen on the previous page, we traverse the list using two different references:
  • \texttt{trav}, which we use as before
  • \texttt{trail}, which stays one node behind \texttt{trav}

```java
StringNode trav = str;
StringNode trail = null;
while (trav != null && trav.ch < ch) {
    trail = trav;
    trav = trav.next;
}
// if trail == null, insert at the front of the list
// else insert after the node to which trail refers
```

Other Variants of Linked Lists

• Doubly linked list

  ```
  \texttt{'c'} \quad \texttt{'a'} \quad \texttt{'t'}
  
  \texttt{prev} \quad \texttt{next}
  
  \texttt{prev} \quad \texttt{next}
  
  \texttt{prev} \quad \texttt{next}
  
  null
  
  null
  
  null
  ```

  • add a \texttt{prev} reference to each node -- refers to the previous node
  • allows us to “back up” from a given node

• Linked list with a dummy node at the front:

  ```
  str \quad \texttt{'c'} \quad \texttt{'a'} \quad \texttt{'t'} \quad null
  
  \texttt{null}
  
  \texttt{null}
  
  \texttt{null}
  ```

  • the dummy node doesn’t contain a data item
  • it eliminates the need for special cases to handle insertion and deletion at the front of the list
  • more on this in the next set of notes