**Pointer review**

- **Pointer/Pointee**
  ```c
  int i;
  int ip = &i;
  ```
- **Dereference**
  ```c
  *ip = 4;
  ```
- **Bad Pointers & segmentation faults**
- **Pointer assignment**
- **Dynamic allocation**
  ```c
  ip = new int(4);
  ```
- **Deallocation: putting back your toys**
  ```c
  delete ip;
  ```
ADTs and vectors, towards linked lists

- `tvector` is a class-based implementation of a lower-level data type called an array
  - `tvector` grows dynamically (doubles in size as needed) when elements inserted with `push_back`
  - `tvector` protects against bad indexing, arrays don’t
  - `tvector` supports assignment: `a = b`, arrays don’t

- As an ADT (abstract data type) vectors support
  - *Constant-time* or O(1) access to the k-th element
  - *Amortized* linear or O(n) storage/time with `push_back`
    - Total storage used in n-element vector is approx. 2n, spread over all accesses/additions (why?)
  - Adding a new value in the middle of a vector is expensive, linear or O(n) because shifting required
Linked lists

- Low-level (concrete) data structure, used to implement higher-level structures
  - Used to implement sequences/lists (see CList in Tapestry)
  - Basis of common hash-table implementations (later)
  - Similar to how trees are implemented, but simpler

- Linked lists as ADT
  - Constant-time or O(1) insertion/deletion from anywhere in list, but first must get to the list location
  - Linear or O(n) time to find an element, sequential search
  - Like a film or video tape: splicing possible, access slow

- Good for sparse structures: when space is scarce, allocate exactly as many list elements as needed
Linked list applications

- Remove element from middle of a collection, maintain order, no shifting. Add an element in the middle, no shifting
  - What’s the problem with a vector (array)?
  - Emacs visits several files, internally keeps a linked-list of buffers
  - Naively keep characters in a linked list, but in practice too much storage, need more esoteric data structures

- What’s \((3x^5 + 2x^3 + x + 5) + (2x^4 + 5x^3 + x^2 + 4x)\) ?
  - As a vector \((3, 0, 2, 0, 1, 5)\) and \((0, 2, 5, 1, 4, 0)\)
  - As a list \(((3,5), (2,3), (1,1), (5,0))\) and ________?
  - Most polynomial operations sequentially visit terms, don’t need random access, do need “splicing”

- What about \((3x^{100} + 5)\) ?
Linked lists, CDT and ADT

- As an ADT
  - A list is empty, or contains an element and a list
  - ( ) or (x, (y, ( ) ) )

- As a picture

- As a CDT (concrete data type)
  ```c
  struct Node
  {
    string info;
    Node * next;
  };
  ```

  Node * p = new Node();
  p->info = "hello";
  p->next = NULL;  // 0
Building linked lists

• Add words to the front of a list (draw a picture)
  ↠ Create new node with next pointing to list, reset start of list
  ```
  struct Node {
    string info;
    Node * next;
    Node(const string& s, Node * link) :
      info(s), next(link) {
    }
  };
  // ... declarations here
  Node * list = 0;
  while (input >> word)
  {
    list = new Node(word, list);
  }
  ```

• What about adding to the end of the list?
Standard list processing (iterative)

- Visit all nodes once, e.g., count them
  - Why does the while loop version work?

```c
int Size(Node * list) {
    int count = 0;
    while (list != 0) {
        count++;
        list = list->next;
    }
    return count;
}
```

- Print nodes?
- Append “s” to all strings in list?
Standard list processing (recursive)

- Visit all nodes once, e.g., count them

```c
int Size(Node * list)
{
    if (list == 0) return 0;
    return 1 + Size(list->next);
}
```

- Base case is almost always empty list – NULL/0 node
  - Must return correct value, perform correct action
  - Recursive calls use this value/state to anchor recursion
  - Sometimes one node list also used, two “base” cases
- Recursive calls make progress towards base case
  - Almost always using list->next as argument
Recursion and linked lists

• Print nodes in reverse order
  ➔ Print all but first node and...
    • Print first node before or after other printing?

```c
void Print(Node * list)
{
    if (list != 0)
    {
        // Code continues here...
    }
}
```
Linked list special cases

● When passing list as parameter, what can be changed?
  ➔ What part is copied and what is shared?
  ➔ How does passing by reference help?

● How to add a node to an empty list?

```cpp
void Insert(Node *& list, const string& key) {
  if (list != 0) {

  }
  else {

  }
}
```

● How to remove the first node of a list?
Changing a linked list recursively

- Alternately, pass in a list, return altered list

```c
list = Change(list, /* other params */);
Node * Change(Node * list, const string& key)
{
    if (list != 0)
    {
        list->next = Change(list->next, key);
        if (list->info == key) return list->next;
        else return list;
    }
    return 0;
}
```

- What does this code do? How can we reason about it?
  ➔ Empty list, one-node list, two-node list, n-node list
  ➔ Similar to proof by induction
Reverse list: (a, b, c, d) to (d, c, b, a)

Node * Reverse(Node * list)
// post: return list reversed - list changed,
//       new nodes NOT created
{
    if (list != 0)
    {

        Node * rest = Reverse(list->next);

    }
    return ;
}

Header (aka dummy) nodes

- Special cases in code lead to problems
  - Permeate the code, hard to reason about correctness
  - Avoid special cases when trade-offs permit
    - Space, time trade-offs

- In linked lists it is useful to have a header node, the empty list is not NULL/0, but a single “blank” node
  - Every node has a node before it, avoid special code for empty lists
  - Header node is skipped by some functions, e.g., count the values in a list
  - What about a special “trailing” node?
  - What value is stored in the header node?
Variations: doubly linked

- In singly-linked lists, need pointer-to-node before to remove a node from a list, why?

  ↗  How do header nodes help?

- Move forward/backwards in a doubly linked list, what needs to be added to Node declaration?
  ↗  Downside?
Circularly linked list

- If the last node points to NULL/0, the pointer is “wasted”
- Can make list circular, so it is easy to add to front or back
  - Want only one pointer to list, should it point at first or last node?
  - How to create first node?
  - Potential problems? Failures?

```c
// circularly linked, list points at last node
Node * first = list->next;
Node * current = first;
do
{    Process(current);
    current = current->next;
} while (current != first);
```
Hybrid structures: vectors and lists

- We can store word/counts in a vector
- We can store word/counts in linked list
  - Advantages of approaches?
  - Alternatives within an approach? Between?

- What about a vector of linked lists?
  - One linked list per letter of the alphabet: ‘a’ – ‘z’
  - Why use vector of linked lists rather than linked list of linked lists?
  - What about a vector of vectors? Possible? Drawbacks?
  - What about more than 26 linked lists, 52? Ten-thousand?