Data Structures for NLP
A Tutorial for NLP (CSE 562/662)

Kristy Hollingshead
Winter 2007

`www.cslu.ogi.edu/~hollingk/NLP_tutorial.html`

Disclaimers

- Your coding experience
  - Tutorial intended for beginners up to experts
- C/C++/Java
  - Examples will be provided in C
  - Easily extended to C++ classes
  - Can also use Java classes, though will be slower—maybe prohibitively so
- compiling C
  - `gcc -Wall foo.c -o foo`
  - `-g` to debug with `gdb`

Overview

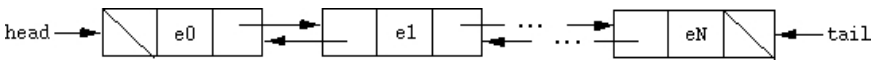
- Storage
  - Lists
  - Trees
  - Pairs (frequency counts)
  - Memory allocation
- Search
  - Efficiency
    - Hash tables
  - Repetition
- Code
  - http://www.cslu.ogi.edu/~hollingk/code/nlp.c

3

Linked Lists (intro)

- for each list:
  - first/head node
  - last/tail node (opt)
- for each node:
  - next node
  - previous node (opt)
  - data
- vs arrays

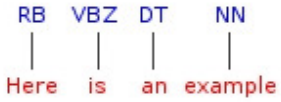
```
struct node;
typedef struct node Node;
typedef struct list {
  Node *head;
  Node *tail;
} List;
struct node {
  char *label;
  Node *next;
  Node *prev
};
```



4

Linked Lists (NLP)

- example: POS sequence
  (RB Here) (VBZ is) (DT an) (NN example)



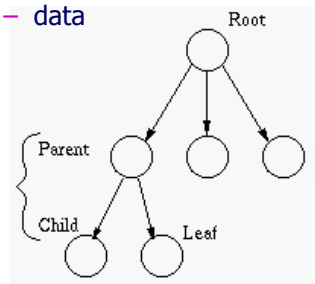
- reading in from text (pseudo-code):

```
read_nodes {
  while curr_char != '\n' {
    if (curr_char=='(') {
      prevnode=node; node=new_node();
      node->prev=prevnode;
      if (prevnode!=NULL) prevnode->next=node; }
    node->pos=read_until(curr_char,' ');
    curr_char++; // skip ' '
    node->word=read_until(curr_char,')');
    curr_char++; // skip ')'
}
```

5

Trees (intro)

- for each tree:
  - root node
  - next tree (opt)
- for each node:
  - parent node
  - children node(s)
  - data

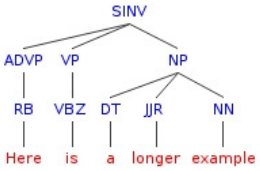



```
struct tree;
typedef struct tree Tree;
struct node;
typedef struct node Node;
struct tree {
  Node* root;
  Tree* next;
};
struct node {
  char* label;
  Node* parent;
  int num_children;
  Node* children[ ];
};
```

6

Trees (NLP)

- Examples:
  - parse trees
    (SINV (ADVP (RB Here)) (VP (VBZ is))
    (NP (DT a) (JJR longer) (NN example)) (. .))
  - grammar productions
    NP => DT JJR NN



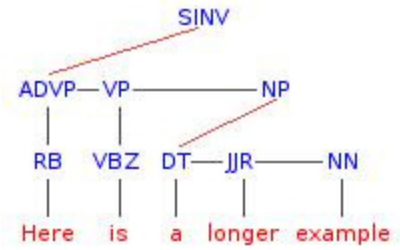
- reading in from text (pseudo-code):

```
read_trees {
  if (curr_char=='(') {
    node=new_node(); node->lbl=read_until(curr_char,' '); }
  if (next_char!='(') node->word=read_until(curr_char,')');
  if (next_char==')') return node; // "pop"
  else node->child=read_trees();   // recurse
}
```

7

Lists *in* Trees (NLP)

- navigation in trees
- convenient to link to "siblings"
  - right sibling ≈ next node
  - left sibling ≈ previous node
- convenient to "grow" children
  - children ≈ first child + right siblings



8

Pairs / Frequency Counts

- Examples
  - What POS tags occurred before this POS tag?
  - What POS tags occurred with this word?
  - What RHS's have occurred with this LHS?
- Lists
  - linear search— only for short lists!
- Counts
  - parallel array
  - or create a 'Pair' data structure!

```
struct pos {
  char *label;
  int numprev;
  struct pos **bitags; }
struct word {
  char *label;
  int numtags;
  struct pos **tags; }
struct rule {
  char *lhs;
  int numrhs;
  struct rhs **rhss; }
struct rhs {
  int len;
  char **labels; }
```

9

Memory allocation

- allocation
  - multi-dimensional arrays (up to 3 dim)
- initialization
  - malloc vs calloc
- re-allocation
  - realloc, re-initialize
- pointers
  - minimize wasted space given sparse data sets
- de-referencing

```
int *i;
i[0] ≈ (*i)
```

```
int **dim2;
dim2=
 malloc(10*sizeof(int));
for (i=0;i<10;i++)
 dim2[i]=
  malloc(20*sizeof(int));
dim2[1][0]=42;
```

```
int *dim1;
dim1=malloc(
  10*20*sizeof(int));
dim1[(1*20)+1]=42;
```

10

Overview

- Storage
  - Lists
  - Trees
  - Pairs (frequency counts)
  - Memory allocation
- Search
  - Efficiency
    - Hash tables
  - Repetition
- Code

11

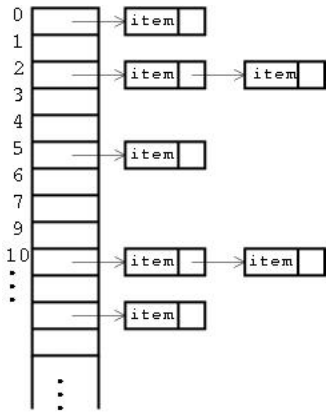
Efficiency

- Huge data sets (productions, tags, features)
  - Efficient data structures
    - structs/classes (vs parallel arrays)
    - hash tables (vs binary sort, qsort, etc.)
- Repetitive, systematic searching
  - Search once, then remember
- Brute force just won't work…

12

Hash Tables (intro)

- Supports efficient look-up (O(1) on avg)
- Maps a key (e.g., *node label*) into a hash code
- Hash code indexes into an array, to find the "bucket" containing desired object (e.g., *node*)
- Collisions
  - Multiple keys (*labels*) mapping to the same "bucket"
  - Chained hashing
  - Open addressing



‡

Chained Hash Table (NLP)

- Data structures‡ to be stored
  - POS data
  - dictionary entries
  - grammar productions
- look-up by label (pseudo-code):

```
typedef struct value {
  char* key;
  int idx;
} Value;

typedef struct hash {
  struct value* v;
  struct hash* next;
} Hash;
```

```
Value* get_value(char* key) {
  int code=get_hash_code(key);
  Value* entry=hash_table[code];
  while (entry && entry->v->key!=key) entry=entry->next;
  if (!entry) make_new_entry(key);
  return entry;
}
```

Repetitious search

- Very repetitive searches in NLP
- Avoid multiple look-ups for the same thing
  - Save a pointer to it
  - Store in a temporary data structure
- Look for patterns
  - Skip *as soon as* you find a (partial) mismatch
    - Make faster comparisons first
      - `(int i == int j)` before `strcmp(s1,s2)`
    - Make "more unique" comparisons first
  - Look for ways to partition the data, save a pointer to each partition
    - Left-factored grammar example

15

Remember...

- Use data structures (structs/classes)
- Allocate memory sparingly
- Efficiency of search is vital
  - Use hash tables
  - Store pointers
- Don't rely on brute force methods

16