STAT 534 Handout 4

April 15, 2003

Array implementation of dynamic sets

©Marina Meilă

mmp@stat.washington.edu

1 What are dynamic sets?

A **dynamic set** is a set that can change (by gaining or losing **elements**) over the execution of a program. It is assumed that the elements of the set contain a **field** by whose value they can be ordered. This field is called the **key**.

Examples: the phone directory (key = last name); a (multidimensional) time-series (key = time); bank accounts (key = account number).

Dynamic sets are an abstract concept, with no direct correspondence in C. But an element of a dynamic set can be implemented in C by a `struct`.

Example: the list of students in this class.

```
struct student_record {
  char _c20_last_name[ 20 ];
  char _c20_first_name[ 20 ];
  char_c4_department[ 4 ];
  int _i_year_birth;
  double _d_GPA;
};
typedef struct student_record STUDENT_REC;
STUDENT_RECSTUDENT_REC_set[ 30 ];
```

Key can be `i_year_birth`, or `_c20_last_name`, or `d_GPA`, etc.

Operations on dynamic sets are of two kinds: **queries** that return information but do not modify the set, and **modifying operations**. The standard set operations are: SEARCH(S, k), INSERT(S, x), DELETE(S, px), MINIMUM(S), MAXIMUM(S), SUCCESSOR(S, px), PREDECESSOR(S, px). (S is a set, k a key, x an element and px a pointer to an existing element.) If there is no answer (for ex. no PREDECESSOR, then the value returned is a symbol called `NIL`. `NIL` plays the role of the empty pointer.

2 Dynamic sets as an array

Now we implement a dynamic set in a C array. If the set has `n_elements` then we assume that they occupy the first `n_elements` elements of the array.

In our example, the elements are `STUDENT_REC`'s. We assume that we have written a function that copies the contents of a `STUDENT_REC` to another `STUDENT_REC`.

```
STUDENT_REC  STUDENT_REC_set[ 30 ];
int n_elements;

void STUDENT_REC_copy( STUDENT_REC  STUDENT_REC_source,
      STUDENT_REC* pSTUDENT_REC_destination);

int search( STUDENT_REC  STUDENT_REC_set[], int n_elements, int i_year_birth ){
  int i;
  for( i = 0; i < n_elements; i++ )
    if( STUDENT_REC_set[ i ]._i_year_birth == i_year_birth )
      break;
  return i;
}
int STUDENT_REC_insert( STUDENT_REC  STUDENT_REC_set[],
      int * pn_elements, STUDENT_REC  STUDENT_REC_new){
  if (pn_elements < NMAX ) {
    STUDENT_REC_copy( STUDENT_REC_new, & STUDENT_REC_set[ *pn_elements ] );
    *pn_elements++;
```

```
    return 0;
  }
  else
    return 1;
}

int delete( STUDENT_REC STUDENT_REC_set[], int *pn_elements, int i_del ){
  int i;
  if ((i_del < NMAX ) && (i_del >= 0 )) {
    for( i = i_del+1; i < * pn_elements; i++ )
      STUDENT_REC_copy( STUDENT_REC_set[ i ], & STUDENT_REC_set[ i - 1 ] );
    *pn_elements--;
    return 0;
  }
  else
    return 1;
}
```

Complexity

- `STUDENT_REC_search`, `STUDENT_REC_delete`, `STUDENT_REC_maximum`, `STUDENT_REC_minimu` `STUDENT_REC_successor`, `STUDENT_REC_predecessor` require $\mathcal{O}(n)$ operations (where n is the number of elements)
- `STUDENT_REC_insert` is $\mathcal{O}(1)$

This is not optimally efficient, especially for many queries. In fact, the most efficient algorithms for this kind of sets are $\mathcal{O}(\log n)$.