École Centrale de Nantes

AlPro

Tutorial 3

3.1 Merge sort

The merge sort is a recursive sorting algorithm :

1. **If the array has only one element, it is already sorted.**
2. **Otherwise, separate the array into two approximately equal parts.**
3. **Recursively sort the two parts with the merge sort algorithm.**
4. **Merge the two sorted arrays into one sorted array.**

```
// merge sort
function merge_sort(array,b,e){
  // array of integer for b the beginning and e the end
  if(size(array)){
    (Integer e1,Integer b1)←split(array,b,e);
    merge_sort(array,b,e1);
    merge_sort(array,b1,e);
    merge(array,b,e1,b1,e);
  }
}

// split
(Integer,Integer) split(array,b,e){
  Integer e1←((e-b)/2)+b;
  Integer b1←e1+1;
  return (e1,b1);
}

// merge
void function(array,b,e1,b1,e){
  Integer c1←b;
  Integer c2←b1;
  int e11=e1;
  while(c1<=e1 || c2<=e){
    if(array[c1]<array[c2]) )
      c1←c1+1;
    else{
      Integer tmp←array[c2];
      for(Integer i=c2-1;i>=c1;i--)
        array[i+1]←array[i];
      array[c1]←tmp;
      c2←c2+1;
      e11++;
    }
  }
}
```

3.2 Processing of machining orders

We consider a machining machine that processes the machining orders in an unsorted order list present in a file. An order contains the type of part, the machining duration (in hours), and the desired end date (in hours from an absolute reference).

— **define a simply linked list structure to manipulate a list of production orders,**

```
typedef struct order{
  char type;
  int duration;
  int end_date;
}order;

typedef struct element{
  order data;
  struct element *next;
}element;
```

3.2.1 Processing in the file order

Initially, the machining orders will be processed in the order of the file. Write the secondary functions and the main program to :

— **load production orders from a file into memory,**
— **process the production orders, with a display of the list and the production delay at each modification,**

```
/* main function */
Integer main(){
  order queue←read_data(path);
  Integer time←0;
  print_order(queue);
  while(queue != NULL){
    time←remove_first_order(queue);
    print(time);
    print_order(queue)
  }return 0;
}

/* read data from a file, with in each line the type, the duration and the
   desired end date of an order */
element* read_data(char* path){
  element list;
  element* current=&list;
  char* line=get_next_line();
  while(not EOF){ // not end of file
    order new;
    new.type←read_next(line);
    new.duration←atoi(read_next(line));
    new.end_date←atoi(read_next(line));
    current->data←new;
    current->next←NULL;
    current←current->next
  }return *list;
}

/* print a list of orders (only their types) */
void print_order(element *list){
  if(list != NULL){
    printf("%c ",list->data.type);
```

```
    print_order(list->next);
  }
}

/* remove the first order of the list */
int remove_first_order(element *list){
  if(list != NULL){
    int time<-list->data.duration;
    list<-list->next;
  }return time;
}
```

3.2.2 Processing in order of finish date

— **In a second step, the machine processes first the production order with the smallest desired end date.**

```
/* main function */
Integer main(){
  order queue<-read_data(path);
  Integer time<-0;
  print_order(queue);
  while(queue != NULL){
    time<-remove_urgent_order(queue);
    print(time);
    print_order(queue)
  }return 0;
}

/* compare end dates and return 1 if more urgent, 0 if equal and -1 if less
   urgent*/
Integer more_urgent(element e1, element e2){
  if(e1.data.end_date < e2.data.end_date) return 1;
  if(e1.data.end_date == e2.data.end_date) return 0;
  return -1;
}

/* return the most urgent order in the list of orders */
element most_urgent(element *list){
  if(list == NULL) return NULL;
  if(list->next == NULL) return list->data;
  element max=most_urgent(list->next);
  if(more_urgent(list->data,max))
    return list->data;
  return max;
}

/* remove a given element if it is in the list of orders (only using end date) */
element* remove_given_order(element *list, element e){
  if(list == NULL) return list;
  if(more_urgent(list->data,e) == 0) return list->next;
  list->next<-remove_given_order(list->next,e);
  return list;
}

/* remove the urgent order of the list */
Integer remove_urgent_order(element *list){
  if(list != NULL){
    element urgent<-most_urgent(list);
```

```
    int time←urgent.data.duration;
    list←remove_given_order(list,urgent);
  }return 0;
}
```