Stuparu Elena Natalia, grupa 412D

Lucrarea 1 SDA (9.04.2021)

Problema 1)

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE      100
typedef int T;
typedef struct ListNode {
   T value;
   struct ListNode* next;
   struct ListNode* prev;
} ListNode;
typedef struct List {
   ListNode* first;
   ListNode* last;
} List;

ListNode *createListNode(T value);
List createList(T value);
List nill(void);
int isEmpty(List list);
List enqueue(List list, T value);
int contains(List list, T value);
List dequeue(List list);
int top(List list);
int length(List list);
List destroyList(List list);
void print(List list);

int main() {
   List l = nill();
```

```
    l = createList(3);
    l = enqueue(l, 4);
    l = enqueue(l, 5);
    l = enqueue(l, 6);
    print(l);
    printf("Va iesi un elementul %d din coada\n", top(l));
    print(l);
    dequeue(l);
    if(contains(l, 4))
        printf("coada contine elementul 4\n");
    destroyList(l);
    return 0;
}
ListNode *createListNode(T value) {
    ListNode node = (ListNode) malloc(sizeof(ListNode));
    node->value = value;
    node->prev = NULL;
    node->next = NULL;
    return node;
}
List nill(void) {
    List list;
    list.first = list.last = NULL;
    return list;
}
List createList(T value) {
    List list;
    list = nill();
    list.first = createListNode(value);
    list.last = list.first;
    return list;
```

```
}
int isEmpty(List list) {
   if(list.first == NULL && list.last == NULL)
      return 1;
   else
      return 0;
}
List enqueue(List list, T value) {
   if(list.first == NULL){
      list = createList(value);
      return list;
   }
   else{
      ListNode *temp;
      ListNode *p = createListNode(value);
      temp = list.last;
      list.last = p;
      temp->next = p;
      p->prev = temp;
      //list.first = p;
      return list;
   }
}
List dequeue(List list) {
   if(isEmpty(list))
      return list;
   else{
      ListNode *temp = list.first;
      list.first->next->prev = NULL;
      list.first = list.first->next;
      free(temp);
```

```
      return list;
   }
}
int top(List list) {
   if(isEmpty(list))
      return -1;
   else
      return list.first->value;
}
int contains(List list, T value) {
   int valid = 0;
   if(list.first == NULL)
      return 0;
   else{
      ListNode *temp = list.first;
      while(temp!= list.last){
         if(temp->value == value)
            valid = 1;
         temp = temp->next;
      }
      if(temp == list.last && temp->value == value)
         valid = 1;
      if(valid)
         return 1;
      else
         return 0;
   }
}
void print(List list) {
   ListNode *temp;
   if(!isEmpty(list)){
```

```
        temp = list.first;
        while(temp != NULL) {
            printf("%d ", temp->value);
            temp = temp->next;
        }
        printf("\n");
    }
}
List destroyList(List list) {
    ListNode *temp, *aux;
    temp = list.first;
    while(temp != list.last){
        aux = temp;
        temp = temp->next;
        list.first = temp;
        temp->prev = NULL;
        free(aux);
    }
    list.first = NULL;
    list.last = NULL;
    free(temp);
    return list;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE    100

typedef int T;
typedef struct ListNode {
    T value;
    struct ListNode* next;
    struct ListNode* prev;
} ListNode;

typedef struct List {
    ListNode* first;
    ListNode* last;
} List;

ListNode *createListNode(T value);
List createList(T value);
List nill(void);
int isEmpty(List list);
List enqueue(List list, T value);
int contains(List list, T value);
List dequeue(List list);
int top(List list);
int length(List list);
List destroyList(List list);
void print(List list);

int main() {
    List l = nill();
    l = createList( value: 3);
    l = enqueue(l,  value: 4);
    l = enqueue(l,  value: 5);
    l = enqueue(l,  value: 6);
    print(l);
    printf( _Format: "Va iesi un elementul %d din coada\n", top(l));
    print(l);
    dequeue(l);
    if(contains(l,  value: 4))
```

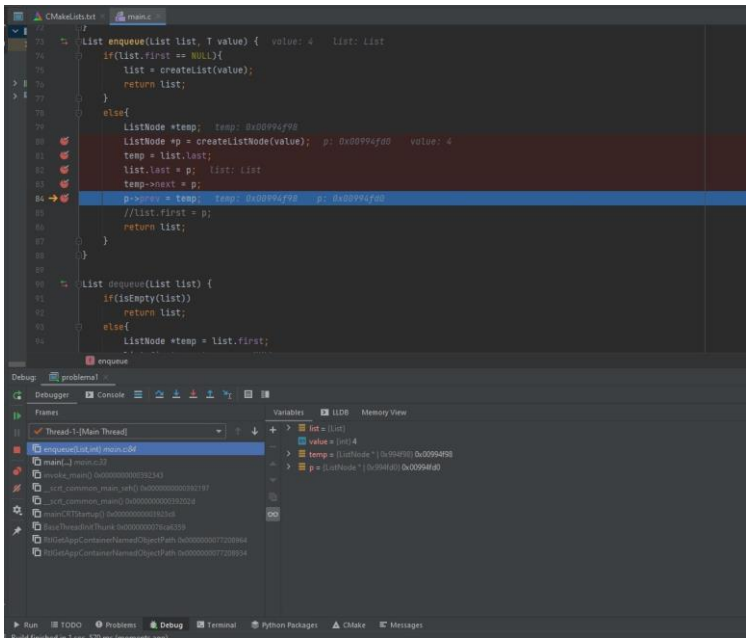
```c
ListNode *createListNode(T value) {
    ListNode *node = (ListNode*) malloc(sizeof(ListNode));
    node->value = value;
    node->prev = NULL;
    node->next = NULL;
    return node;
}

List nill(void) {
    List list;
    list.first = list.last = NULL;
    return list;
}

List createList(T value) {
    List list;
    list = nill();
    list.first = createListNode(value);
    list.last = list.first;
    return list;
}


int isEmpty(List list) {
    if(list.first == NULL && list.last == NULL)
        return 1;
    else
        return 0;
}
List enqueue(List list, T value) {
    if(list.first == NULL){
        list = createList(value);
        return list;
    }
    else{
        ListNode *temp;
        ListNode *p = createListNode(value);
```

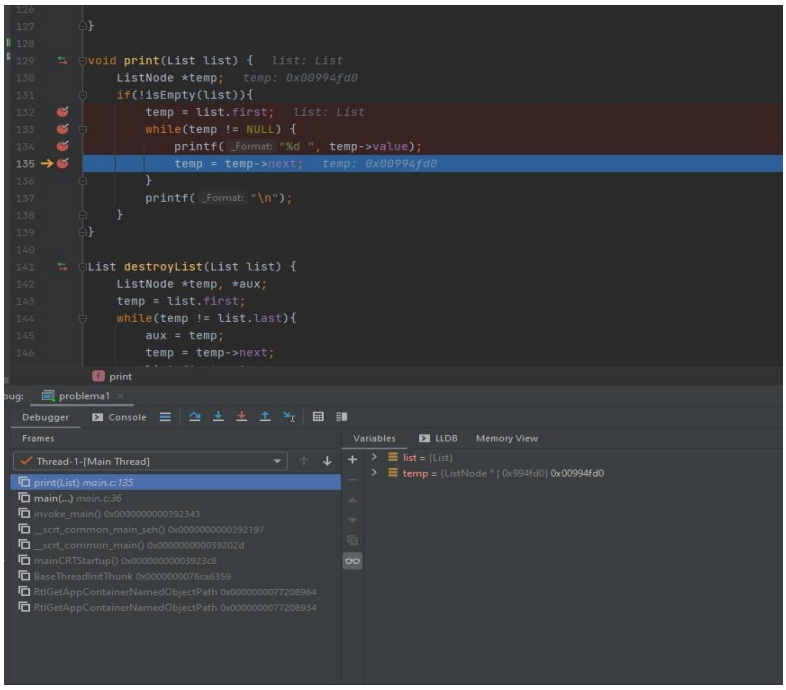
```
List enqueue(List list, T value) {
    if(list.first == NULL){
        list = createList(value);
        return list;
    }
    else{
        ListNode *temp;
        ListNode *p = createListNode(value);
        temp = list.last;
        list.last = p;
        temp->next = p;
        p->prev = temp;
        //list.first = p;
        return list;
    }
}

List dequeue(List list) {
    if(isEmpty(list))
        return list;
    else{
        ListNode *temp = list.first;
        list.first->next->prev = NULL;
        list.first = list.first->next;
        free(temp);
        return list;
    }
}
int top(List list) {
    if(isEmpty(list))
        return -1;
    else
        return list.first->value;
}
```

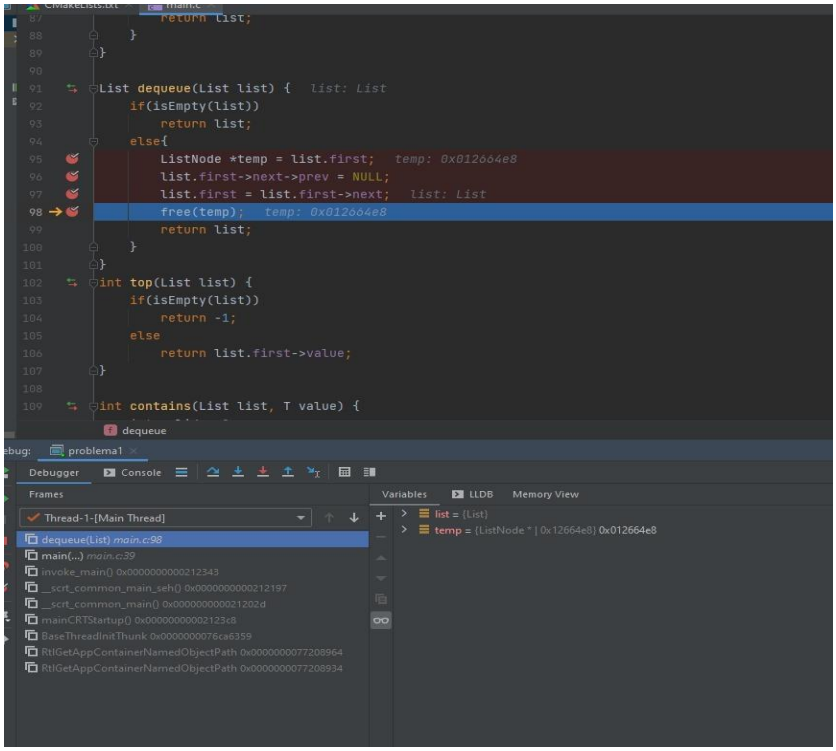
```
int contains(List list, T value) {
    int valid = 0;
    if(list.first == NULL)
        return 0;
    else{
        ListNode *temp = list.first;
        while(temp!= list.last){
            if(temp->value == value)
                valid = 1;
            temp = temp->next;
        }
        if(temp == list.last && temp->value == value)
            valid = 1;
        if(valid)
            return 1;
        else
            return 0;
    }

}

void print(List list) {
    ListNode *temp;
    if(!isEmpty(list)){
        temp = list.first;
        while(temp != NULL) {
            printf( _Format: "%d ", temp->value);
            temp = temp->next;
        }
        printf( _Format: "\n");
    }
}

List destroyList(List list) {
    ListNode *temp, *aux;
    temp = list.first;
```

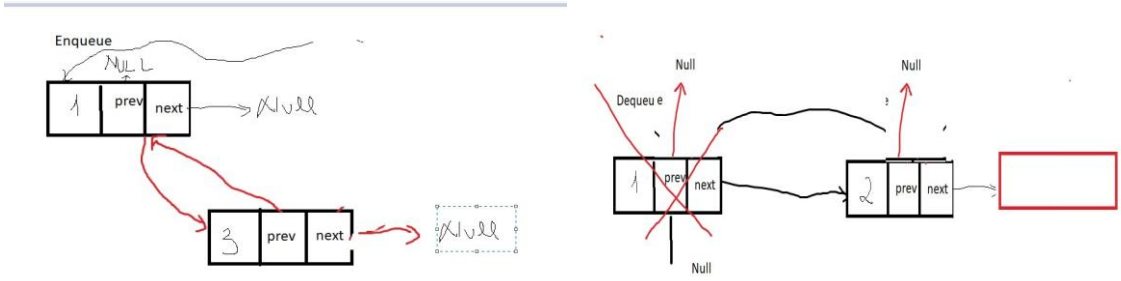
```
List destroyList(List list) {
    ListNode *temp, *aux;
    temp = list.first;
    while(temp != list.last){
        aux = temp;
        temp = temp->next;
        list.first = temp;
        temp->prev = NULL;
        free(aux);
    }
    list.first = NULL;
    list.last = NULL;
    free(temp);
    return list;
}
```

Listare functii:









In momentul crearii liste vom pune primul element in lista (createListNode) si vom atribui valoare null pentru urmatorul si predecesorul. Primul si ultimul element sunt nule. In functia de adugare in coada (enqueu) vom intreba daca lista e goala. In acest caz, return o noua lista cu elementul pe primul loc; Altfel vom folosi variaila temp pentru a ne ajuta sa tinem minte pointerul spre ultimul element,. Apoi adaugam in coada pe ultima pozitie elemntul dorit si refacem legaturile cu variabila temp. Pentru functia top aceasta doar intoarce primul element din coada. In cadrul functiei contains am folosit o parcurgere [prin lista pana la elementul dorit si in cazul in care l-am gasit (temp->value == value) ne oprim si vom intoarce 1. De asemenea, si functia de listare (print) are o parcurgere prin lista folosind variabila temp si afisind fiecare element in parte.

Problema 2)

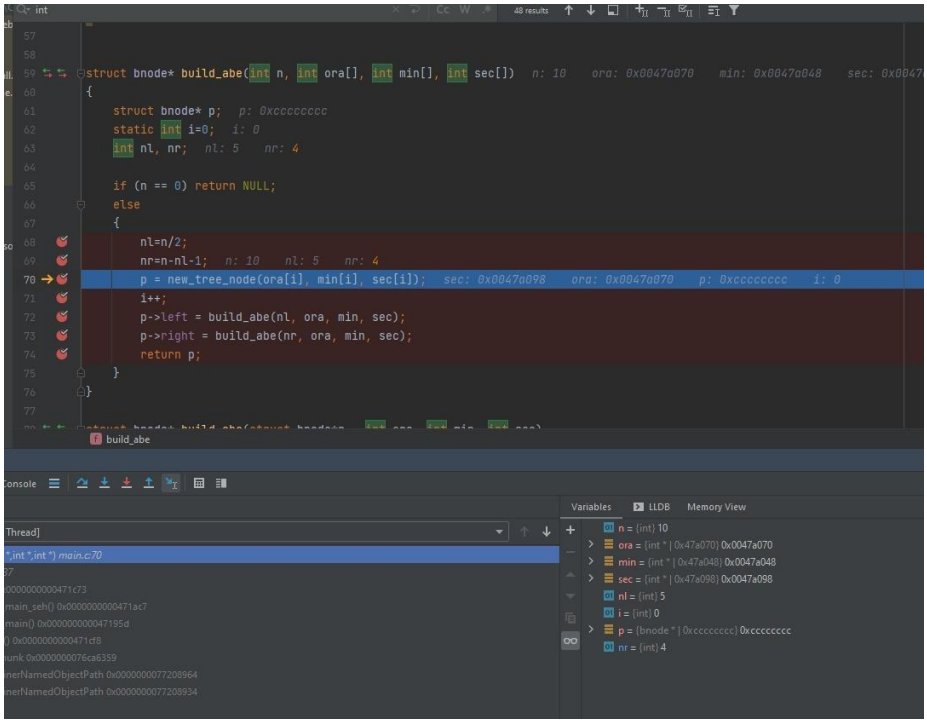
```
#include <stdio.h>

#include <stdlib.h>

struct bnode {

    int ora;

    int min;
```

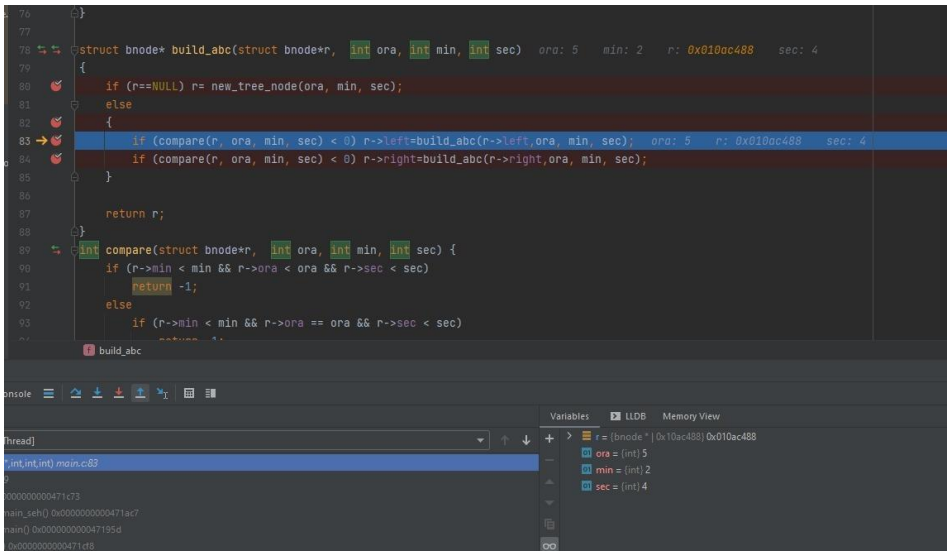
```
    int sec;
    struct bnode* left;
    struct bnode* right;
};
struct bnode* new_tree_node(int ora, int min, int sec);
int compare(struct bnode*r,  int ora, int min, int sec);
struct bnode* build_abe(int n, int ora[], int min[], int sec[]);

struct bnode* build_abc(struct bnode*r, int ora, int min, int sec);
struct bnode* search_abc(struct bnode*r, int ora, int min, int sec);

void ldr(struct bnode* r);
void dlr(struct bnode* r);
void lrd(struct bnode* r);
int sir[10]={1,2,3,4,5,6,7,8,9,10};
int sir1[10]={3,5,10,7,9,8,2,6,1,4};
int sir2[10]={3,4,10,10,9,8,3,6,5,4};
int main() {
  int i;
  struct bnode* roote = NULL;
  struct bnode* rootc = NULL;

 roote=build_abe(10,sir1, sir, sir2);
    for (i=0; i < 10; i++)
        rootc=build_abc(rootc,sir1[i], sir[i], sir2[i]);
    ldr(rootc);
    printf("\n");
   return 0;
}
struct bnode* new_tree_node(int ora, int min, int sec)
{
```

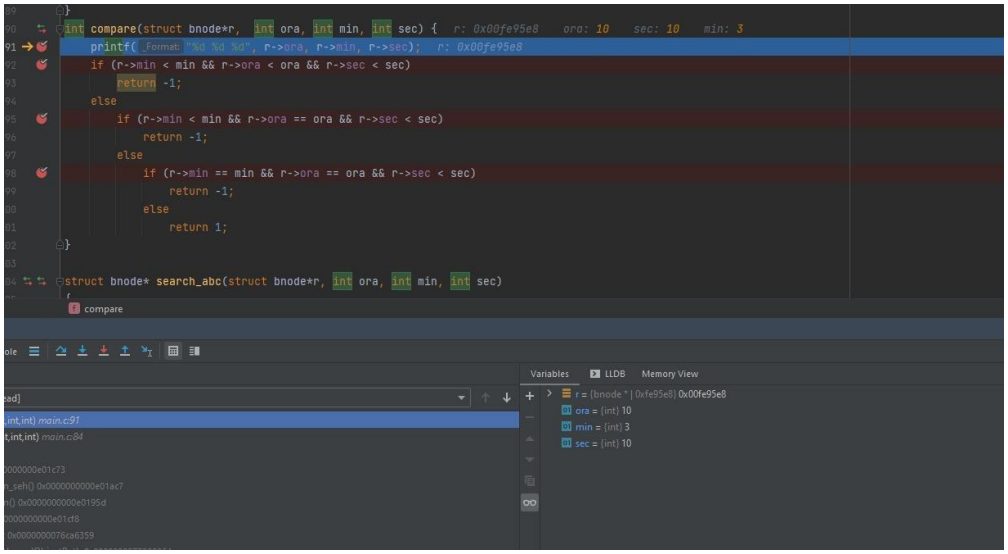
```
    struct bnode* p;
   p= (struct bnode*) malloc(sizeof(struct bnode));
    p->min=min;
    p->ora = ora;
    p->sec = sec;
    p->left=NULL;
    p->right=NULL;
}
struct bnode* build_abe(int n, int ora[], int min[], int sec[])
{
    struct bnode* p;
    static int i=0;
    int nl, nr;
   if (n == 0) return NULL;
    else
    {
       nl=n/2;
       nr=n-nl-1;
       p = new_tree_node(ora[i], min[i], sec[i]);
       i++;
       p->left = build_abe(nl, ora, min, sec);
       p->right = build_abe(nr, ora, min, sec);
       return p;
    }
}
struct bnode* build_abc(struct bnode*r,  int ora, int min, int sec)
{
    if (r==NULL) r= new_tree_node(ora, min, sec);
    else
    {
       if (compare(r, ora, min, sec) < 0) r->left=build_abc(r->left,ora, min, sec);
```

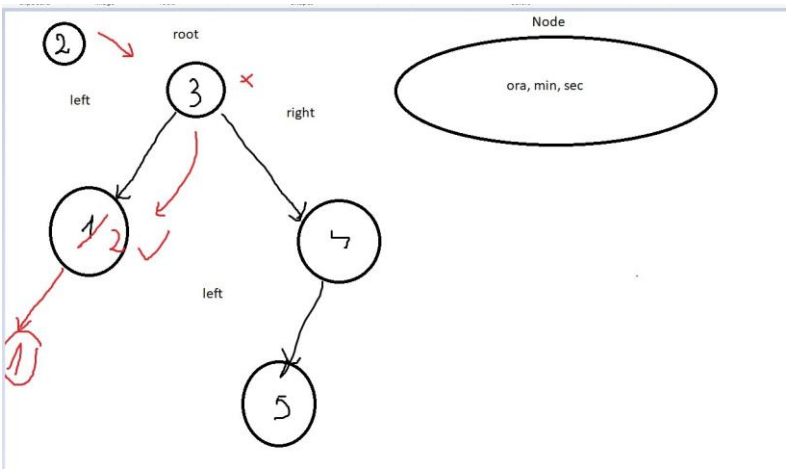
```
        if (compare(r, ora, min, sec) < 0) r->right=build_abc(r->right,ora, min, sec);
    }
 return r;
}
int compare(struct bnode*r,  int ora, int min, int sec) {
    printf("%d %d %d", r->ora, r->min, r->sec);
    if (r->min < min && r->ora < ora && r->sec < sec)
        return 1;
    else
        if (r->min < min && r->ora == ora && r->sec < sec)
            return 1;
        else
            if (r->min == min && r->ora == ora && r->sec < sec)
                return 1;
            else
                return -1;
}
struct bnode* search_abc(struct bnode*r, int ora, int min, int sec)
{
    if (r == NULL)  return NULL;
    if (r->min == min && r->ora == ora && r->sec == sec) return r;
    if (compare(r, ora, min, sec) < 0) return(search_abc(r->left, ora, min, sec));
    if (compare(r, ora, min, sec) > 0) return (search_abc(r->right, ora, min, sec));
}
void ldr(struct bnode* r)
{
    if(r!=NULL)
    {
        ldr(r->left);
        printf("%d %d %d, ", r->ora, r->min, r->sec);
        ldr(r->right);
```

```
    }
}
void dlr(struct bnode* r)
{
    if(r!=NULL)
    {
        printf("%d %d %d, ", r->ora, r->min, r->sec);
        dlr(r->left);
        dlr(r->right);
    }
}
void lrd(struct bnode* r)
{
    if(r!=NULL)
    {
        lrd(r->left);
        lrd(r->right);
        printf("%d %d %d, ", r->ora, r->min, r->sec);
    }
}
```


struct bnode* build_abe(int n, int ora[], int min[], int sec[]) n: 10 ora: 0x0047a070 min: 0x0047a048 sec: 0x0047
{
struct bnode* p; p: 0xcccccccc
static int i=0; i: 0
int nl, nr; nl: 5 nr: 4
if (n == 0) return NULL;
else
{
nl=n/2;
nr=n-nl-1; n: 10 nl: 5 nr: 4
p = new_tree_node(ora[i], min[i], sec[i]); sec: 0x0047a098 ora: 0x0047a070 p: 0xcccccccc i: 0
i++;
p->left = build_abe(nl, ora, min, sec);
p->right = build_abe(nr, ora, min, sec);
return p;
}
}
build_abe
Variables
LLDB
Memory View
n = {int} 10
ora = {int * | 0x47a070} 0x0047a070
min = {int * | 0x47a048} 0x0047a048
sec = {int * | 0x47a098} 0x0047a098
nl = {int} 5
i = {int} 0
p = {bnode * | 0xcccccccc} 0xcccccccc
nr = {int} 4


}
struct bnode* build_abc(struct bnode*r, int ora, int min, int sec) ora: 5 min: 2 r: 0x010ac488 sec: 4
{
if (r==NULL) r= new_tree_node(ora, min, sec);
else
{
if (compare(r, ora, min, sec) < 0) r->left=build_abc(r->left,ora, min, sec); ora: 5 r: 0x010ac488 sec: 4
if (compare(r, ora, min, sec) < 0) r->right=build_abc(r->right,ora, min, sec);
}
return r;
}
int compare(struct bnode*r, int ora, int min, int sec) {
if (r->min < min && r->ora < ora && r->sec < sec)
return -1;
else
if (r->min < min && r->ora == ora && r->sec < sec)
build_abc
Variables
LLDB
Memory View
r = {bnode * | 0x10ac488} 0x010ac488
ora = {int} 5
min = {int} 2
sec = {int} 4


}
int compare(struct bnode*r, int ora, int min, int sec) { r: 0x00fe95e8 ora: 10 sec: 10 min: 3
printf(_Format "%d %d %d", r->ora, r->min, r->sec); r: 0x00fe95e8
if (r->min < min && r->ora < ora && r->sec < sec)
return -1;
else
if (r->min < min && r->ora == ora && r->sec < sec)
return -1;
else
if (r->min == min && r->ora == ora && r->sec < sec)
return -1;
else
return 1;
}
struct bnode* search_abc(struct bnode*r, int ora, int min, int sec)
compare
Variables
LLDB
Memory View
r = {bnode * | 0xfe95e8} 0x00fe95e8
ora = {int} 10
min = {int} 3
sec = {int} 10



Fiecare nod este descris prin ora, min, sec. Daca dorim sa adugam elementul nodul 2 (notatie speciala), vom incepe de la root, vom compara valorile din root cu cele din nodul 2 folosind functia compare. In momentul acesta functia compare va intoarce -1 si ne indreptam spre left. Ajungem in nodul 1, iar raspunsul functiei compare va fi 1. Atunci vom adauga aici nodul 2 si vom pune nodul 1 ca fiind copilul stang al acestuia. Pe acelasi principiu este construita si functia seach_abc asa cum am descris in diagrama. Functia de compare va returna 1 daca rootul este mai mic si astfel se va merge pe partea dreapta. Altfel va intoarce -1.