1 Lösungsidee

Für diese Übung soll die WorkManagementUnit der Übung 7 umgebaut werden. Hierbei ist besonders darauf zu achten, dass die Schnittstelle für den Aufrufer sich nicht ändert.
Es soll lediglich das Gedächtnis durch einen Baum ausgetauscht werden.

Bei den bereits spezifizierten Datentypen für die Personen sollen die Attribute next, prev in left, right umbenannt werden, da die alten Bezeichnungen bei einen Baum nicht mehr zutreffenden sind. Des Weiteren soll der Datentyp PersonItemList in PersonItemTree umbenannt werden, da auch hier die Bezeichnung nicht mehr zutreffend ist. Ebenso soll mit der globalen Variable ‚list' verfahren werden. Prozeduren und Funktionen deren Namen nicht mehr zutreffend sind sollen ebenso angepasst werden. Alle diese Änderungen sollen sich lediglich auf die privaten teile der Unit beschränken. Hilfsprozeduren und –Funktionen sollen nach eigenen ermessen implementiert werden um das Arbeiten mit dem Baum zu erleichtern.

Das Ausbalancieren des Baums soll bereits beim Hinzufügen einer neuen Person angewendet werden, da bei jedem Speichern eines Eintrages nach der Person gesucht wird, ob diese bereits einen Eintrag hat. Das Ausbalancieren soll hierbei alle 100 Personen erfolgen, da anzunehmen ist, das bis 100 Personen im Baum, welcher möglicherweise außer Balance ist, keine nennenswerten Performanceeinbußen zu erwarten sind.
Das Ausbalancieren soll implementiert werden, um einerseits die Rekursionstiefe gering zu halten und andererseits um die Suche nach einer bestimmten Person zu optimieren.
Hierbei soll aber der Baum nur dann ausbalanciert werden, wenn der Baum vorherig als nicht balanciert markiert ist. Dies bedeutet, dass der Baum als nicht balanciert gilt, sobald ein neuer Eintrag dem Baum hinzugefügt wurde. Dies soll so gelöst werden, das das balancieren des Baums als aufwendig anzusehen ist. Daher kein balancieren des Baums beim Hinzufügen eines neuen Eintrags, sondern nur wenn Prozeduren aufgerufen werden, wo nach Personen gesucht wird.

<u>Algorithmus Ausbalancieren des Baums:</u>

1. Balanciere den Baum aus wenn .n neue Elemente hinzugefügt wurden.
2. Kopiere die Element des Baums sortiert in ein dynamisches Array. (IN-ORDER)
3. Rufe eine rekursive Prozedur auf, die sich wie folgt verhalten soll.
    a. Ermittle das mittlere Element des gegebenen Bereichs.
    b. Füge diese Element dem Baum hinzu, sofern eines zu ermitteln war.
    c. Ermittle die Bereichsgrenzen der linken Hälfte
    d. Rufe für das Setzen des linken Knotens die Prozedur erneut auf mit den neu ermittelten Bereichsgrenzen

e. Ermittle die Bereichsgrenzen der rechten Hälfte
f. Rufe für das Setzen des rechten Knotens die Prozedur erneut auf mit den neu ermittelten Bereichsgrenzen
g. Wiederhole dies bis alle Elemente des Arrays dem Baum hinzugefügt wurden.

Berechnung der aktuellen Bereichsgrenzen:

1. Linke Bereichshälfte:
   a. Linke Grenze = aktuelle linke Grenze
   b. Rechte Grenze = aktueller Index des Baums – 1
2. Rechte Bereichshälfte:
   a. Linke Grenze = aktueller Index des Baums + 1
   b. Rechte Grenze = aktuelle rechte Grenze

Sollten die linke und rechte Bereichsgrenze gleich sein, so liefere das Element an diesem Index.
Sollte die linke Bereichshälfte die rechte Bereichsgrenze überschneiden oder die Differenz der beiden Bereichsgrenzen 0 sein, so gibt es kein Element zu ermitteln.
Sollte die Differenz der beiden Bereichsgrenzen MOD 2 ungleich 0 sein so erhöhe den Index um 1. Dies ist notwendig, da es bei als Bsp.: 4 Elementen keine eindeutige Mitte gibt.
Bsp.:
5 DIV 2 = 2 => 2 + 1 = 3 (Index des mittleren Elementes) | A B **C** D E
4 DIV 2 = 2 (Index mittleren Elementes) | A **B** C D
(Da es hier keine genaue Mitte gibt, ist dieses Ergebnis der Kompromiss)

Die Arbeitseinträge einer Person sollen sortiert in die Liste gespeichert werden. Die Sortierung soll aufsteigend erfolgen. Bei einer großen Anzahl von Einträgen könnte die Performance leiden, da im schlimmsten Fall über die ganze Liste iteriert *werden* muss.

2 Source

Folgend sind die Sources der WorkManagementUnit und deren Tests angeführt.

2.1 WorkManagementUnit

Folgend ist der Source der WorkManagementUnit angeführt.

```
{
  Unit for handling work entries which are hold by a binary tree.
  Call Reset to reset the memory.
  This unit can handle as much entries as the caller wishes, depends on the
  available memory.
}
UNIT WorkManagementUnit;

{ ######################### Interface part ######################### }
INTERFACE

{ Uses the time span unit for time span handling }
USES TImeSpanUnit;

{
  The types which are visible and usable for the caller.
}
TYPE
  { The error codes which can be handled by the caller }
  ErrorCode = (NONE, TO_SHORT, TO_LONG, INVALID_SPAN);

  { Compound representing a work entry, which is only used by the caller for adding a entry }
  WorkEntry = RECORD
    name: STRING;
    task: STRING;
    spendTime: TimeSpan;
  END;

{
  Resets the memory of this module by disposing all elements of the list and also disposing
all elements
  of any contained list of the elements.
}
PROCEDURE Reset;

{
  Creates a work entry for the given data.

  @param
      name: the name of the person
  @param
      task: the name of the task the person worked on
  @param
      spendTime: the TimeSpan instance which represents the time the person has worked
      on the defined task
  @return
      the create workEntry instance
}
FUNCTION CreateWorkEntry(name, task: STRING; spendTime: TimeSpan): WorkEntry;

{
  Adds a WorkEntry to the backed list.

  @param
      entry: the entry to be added
  @param
      error: the error parameter which will be set <> NONE if an error occurs
  @returns
      the occurred error or the given error from the caller which should be NONE
}
PROCEDURE AddWorkEntry(entry: WorkEntry; VAR error: ErrorCode);

{
  Gets the total work time for the given person.

  @param
```

```
      name: the name of the person
  @param
      spendTime: the TimeSpan instance given by the caller which will get set with the total
work time,
      should be initialized by the caller with 0:0:0
  @return
      the TimeSpan representing the total work time, or all values set to 0 when the person
would not be found,
      or no entry exists
}
PROCEDURE GetTotalWorkTimeForPerson(name: STRING; VAR spendTime: TimeSpan);

{
  Gets the average time worked on a task.

  @param
      task: the task to get the average work time for
  @param
      average: the TimeSpan instance where the result will be saved, shall be initialized by
the
      caller with 0:0:0
  @return
      the TimeSpan instance which contains the average time worked on the task, or all field
set to 0,
      if the task could not be found, because there can no task be present in the entry holder
which has
      a time spend less than 1 minute.
}
PROCEDURE GetAverageWorkTimeForTask(task: STRING; VAR average: TimeSpan);

{
  Prints the persons which has worked on the task, along with their time spend.

  @param
      task: the task to be printed.
}
PROCEDURE PrintPersonForTask(task: STRING);

{
  Prints the work summary for the given person.

  @param
      name: the name of the persons to search for the work summary
}
PROCEDURE PrintWorkSummaryForPerson(name: STRING);

{
  Gets the name of the person with the most time spent on the saved tasks.

  @return
      the name of the busiest person
}
FUNCTION BusiestPerson: STRING;

(*
!!!! This function is not supported by this application because not required by the exercise 9
!!!!
{
  Removes all saved entries of a person.

  @param
      name: the name of the person
  @return
      true if the person has been removed, false otherwise
}
FUNCTION DeletePerson(name: STRING): BOOLEAN;
*)

{
  Gets the count of all backed work entries.

  @return
      the count of all work entries
}
FUNCTION GetTotalWorkEntryCount: LONGINT;
```

```
{ ######################## Implementation part ######################## }
IMPLEMENTATION

TYPE
  { Pointer to a WorkItemNode }
  WorkItemNode = ^WorkItem;
  { Compound representing a work item for the person }
  WorkItem = RECORD
    task: STRING;
    spendTime: TimeSpan;
    next: WorkItemNode;
  END;
  { Pointer to a tree of WorkItemNode }
  WorkItemList = WorkItemNode;

  { Pointer to a PersonItem }
  PersonItemNode = ^PersonItem;
  { Compound representing the person in the backed tree }
  PersonItem = RECORD
    name: STRING;
    { WrokItemList with all work items related to this person }
    workItems: WorkItemList;
    left, right: PersonItemNode;
  END;
  { Pointer to the tree of PersonItem }
  PersonItemTree = PersonItemNode;

  { The array of person nodes, used for the dynamic array }
  PersonItemNodeArray = ARRAY OF PersonItemNode;
  { Integer array used for already visited indexes, length will depend on set size }
  IntegerArray = ARRAY OF LONGINT;
  { Pointer to the visited indexes array }
  VisitedIndexesPtr = ^IntegerArray;

VAR
  {
    The memory which contains the persons saved in the system.
    Implemented as a double linked cyclic tree with anchor
  }
  tree: PersonItemTree;
  {
    Keep the size of the save person nodes in the tree,
    so that there will be no need to iterate over the whole tree to determine the saved
element count
  }
  size: LONGINT;

CONST
  {
    Factor which specifies on how many inserted elements in the tree, the tree is
    supposed to be balanced.
    Used because balancing on every insert will decrease performance when many elements
    are present in the tree.
  }
  factor: INTEGER = 100;

{ ######################### Private Function/Procedure part ######################### }
{
  Creates a WorkItemNode instance.

  @param
      entry the WorkEntry instance to create the WorkItemNode instance for
  @return
      the created WorkItemNode instance
}
FUNCTION CreateWorkItemNode(entry: WorkEntry): WorkItemNode;
VAR
  node: WorkItemNode;
BEGIN
  New(node);
  node^.task := LowerCase(entry.task);
  node^.spendTime := entry.spendTime;
  node^.next := NIL;
  CreateWorkItemNode := node;
END;
```

```
{
  Creates a PersonItemNode instance.

  @param
      entry: the entry to create PersonItem instance for
  @return
      the created PersonItemNode instance
}
FUNCTION CreatePersonItemNode(entry: WorkEntry): PersonItemNode;
VAR
  node: PersonItemNode;
BEGIN
  New(node);
  node^.name := LowerCase(entry.name);
  node^.workItems := CreateWorkItemNode(entry);
  node^.left := NIL;
  node^.right := NIL;
  CreatePersonItemNode := node;
END;

{
  Adds a WorkitemNode sorted to the person backed list.

  @param
      person: the person to add the WorkItemNode instance to its backed workItem list
  @param
      entry: the entry which contains the information for the WorkItemNode instance
}
PROCEDURE AddWorkItemSorted(person: PersonItemNode; entry: WorkEntry);
VAR
  pred, succ, node: WorkItemNode;
BEGIN
  pred := NIL;
  succ := person^.workItems;
  node := CreateWorkItemNode(entry);

  { search for succ which has lower task value }
  WHILE ((succ <> NIL) AND (succ^.task <= node^.task)) DO BEGIN
    pred := succ;
    succ := succ^.next;
  END;

  { Add to begin of list }
  IF (pred = NIL) THEN BEGIN
    person^.workItems := node;
  END
  { Add in the middle or end of list }
  ELSE BEGIN
    pred^.next := node;
  END;
  node^.next := succ;
END;

{
  Disposes all WorkItemNode of the given WorkItemList instance.

  @param
      workItemList: the list of WorkItemNode instances
  @return
      the disposed WorkitemList which is being to be NIL
}
PROCEDURE DisposeWorkItemList(VAR workItemList: WorkItemList);
VAR
  next: WorkItemNode;
BEGIN
  {
    IF (workItemList = NIL) THEN BEGIN
      WriteLn('No WorkItems to dispose');
    END;
  }
  WHILE (workItemList <> NIL) DO BEGIN
    next := workItemList^.next;
    {
      WriteLn('Dispose WorkItem: ', workItemList^.task:15, ' | spendTime: ',
TimeSpanUnit.TimeSpanToString(workItemList^.spendTime));
    }
```

```
    Dispose(workItemList);
    workItemList := next;
  END;
END;

{
  Finds the WorkItems of the given PersonItemNode instance for the given task.
  ATTENTION: The returned list needs to be disposed manually.

  @param
      person: the PersonItemNode instance to search for the given task on its WorkItemList
  @param
      task: the task to be searched
  @return
      the created WorkItemList instance if at least one entry for the task could be found, NIL
otherwise
}
FUNCTION FindWorkItemNodesForTask(person: PersonItemNode; task: STRING): WorkItemList;
VAR
  node, newNode: WorkItemNode;
  itemList: WorkItemList;
  lowerTask: STRING;
BEGIN
  itemList := NIL;
  lowerTask := LowerCase(task);

  node := person^.workItems;
  WHILE (node <> NIL) DO BEGIN
    IF (node^.task = lowerTask) THEN BEGIN
      { Clone WorkItem node and put into temp list}
      New(newNode);
      newNode^.task := node^.task;
      newNode^.spendTime := node^.spendTime;
      newNode^.next := NIL;
      IF (itemList = NIL) THEN BEGIN
        itemList := newNode;
      END
      ELSE BEGIN
        newNode^.next := itemList;
        itemList := newNode;
      END;
    END;
    node := node^.next;
  END;

  FindWorkItemNodesForTask := itemList;
END;

{
  Answers the question if the given PersonItemNode instance contains at least one WrokItemNode
instance within its workItems
  for the given task.

  @param
      person: the PersonItemNode instance to search for the given task on its workItems
  @param
      task: the task to search for
  @retrun
      true if the PersonItemNode instance contains at least one WorkItemNode instance with the
given task defined, false otherwise
}
FUNCTION PersonHasTask(person: PersonItemNode; task: STRING): BOOLEAN;
VAR
  node: WorkItemNode;
  lowerTask: STRING;
BEGIN
  lowerTask := LowerCase(task);

  node := person^.workItems;
  WHILE ((node <> NIL) AND (node^.task <> lowerTask)) DO BEGIN
    node := node^.next;
  END;
  PersonHasTask := (node <> NIL);
END;

{
```

```
  Gets the sum of all TimeSpan instances hold by the WorkItemNodes and also the count of
WorkItem in the list

  @param
      itemList: the list of WorkItemNode to sum their backed TimeSpan instances
  @param
      the seconds to build the summary
  @param
      the count to increase by one for each WorkItemNode
  @return
      the sum of all TimeSpan instances and the count of entries of the given WorkItemList.
}
PROCEDURE SumSpendTimeOfWorkItemList(itemList: WorkItemList; VAR seconds: LONGINT; VAR count:
LONGINT);
BEGIN
  WHILE (itemList <> NIL) DO BEGIN
    seconds := seconds + TimeSpanUnit.TimeSpanToSeconds(itemList^.spendTime).timeInSeconds;
    Inc(count);
    itemList := itemList^.next;
  END;
END;

{ ######################### Tree helper function/procedures ######################### }
{
  Creates a PersonItemNodeArray which is filled with the PersonItemNodes, sorted by their
name.

  @return
      the filled PersonItemNodeArray
}
FUNCTION GetSortedPersonItemNodes:PersonItemNodeArray;
  {
    Fills the given array with the sorted PersonItemNodes.
    This is an IN-ORDER recursive implementation.

    @param
        VAR tree: the tree to be traversed
    @param
        VAR personArrayPtr: the array pointer to fill with the persons
    @param
        VAR i: the index for the PersonItemNodeArray
  }
  PROCEDURE FillPersonItemArray(VAR tree: PersonItemTree; VAR personArray:
PersonItemNodeArray; VAR i: LONGINT);
  BEGIN
    IF (tree <> NIL) THEN BEGIN
      FillPersonItemArray(tree^.left, personArray, i);
      Inc(i);
      personArray[i] := tree;
      FillPersonItemArray(tree^.right, personArray, i);
    END;
  END;

VAR
  personArray: PersonItemNodeArray;
  subTree: PersonItemTree;
  i: LONGINT;
BEGIN
  i := 0;
  subTree := tree;

  { Get dynamic array }
  SetLength(personArray, (size + 1));
  { FIll array with sorted person nodes }
  FillPersonItemArray(subTree, personArray, i);
  {
    WriteLn(' ---------- Filled dynamic array BEGIN ---------------');
    WriteLn('Tree size: ', size);
    FOR i := 1 TO size DO BEGIN
      WriteLn('Array node at: idx: ', i, ': ', personArray[i]^.name);
    END;
    WriteLn(' ---------- Filled dynamic array END -----------------');
  }
  GetSortedPersonItemNodes := personArray;
END;
```

```
{
  Inits a IntegerArray with the size of the backed size member.

  @return
      the created IntegerArray instance with the backed size and all elements set to -1
      which marks the elements as not used.
}
FUNCTION InitVisitedArray: IntegerArray;
VAR
  visited: IntegerArray;
  i: LONGINT;
BEGIN
  SetLength(visited, size + 1);
  FOR i := 1 TO size DO BEGIN
    visited[i] := -1;
  END;
  InitVisitedArray := visited;
END;

{
  Answers the question if all of the indexes have been visited.
  This determined by the given IntegerArray where all Elements must have an value
  not equal to -1.

  @param
      visited: the array to search for visited indexes
  @return
      true if all of the saved indexes are not equal to -1, false otherwise
}
FUNCTION AllIndexesVisited(visited: IntegerArray): BOOLEAN;
VAR
  i, temp: LONGINT;
BEGIN
  i := 1;
  temp := -1;
  { Search for already visited index in the array }
  WHILE ((i <= High(visited)) AND (temp <> -1)) DO BEGIN
    temp := visited[i];
    Inc(i);
  END;
  { Check if all indexes have been visited }
  AllIndexesVisited := (temp <> -1);
END;

{
  Calculates the index of middle element.
}
FUNCTION GetMiddleIndex(leftBorder, rightBorder: LONGINT): LONGINT;
VAR
  range, idx: LONGINT;
BEGIN
  idx := 0;
  { If there is no middle element contained in the given borders }
  IF ((leftBorder = 0) OR (rightBorder = 0) OR (leftBorder > rightBorder)) THEN BEGIN
    idx := -1;
  END
  { If the borders have same value then the index of the border is the element }
  ELSE IF (leftBorder = rightBorder) THEN BEGIN
    idx := leftBorder;
  END
  { Calculate the □ex of the middle element }
  ELSE BEGIN
    range := rightBorder - leftBorder + 1;
    IF ((range MOD 2) <> 0) THEN BEGIN
      idx := idx + 1;
    END;
    idx := idx + leftBorder + (range DIV 2);
  END;
  GetMiddleIndex := idx;
END;

{
  Balances the backed tree.
  It is assumed that the curIdx points to a tree which represents the current middle element
of the defined range.
```

```
  @param
      VAR visited: the IntegerArray which holds the already visited indexes
   @param
      VAR personArray: the array which contains all Elements to be added to the tree
   @param
      VAR tree: the current tree to get its left and right child
   @param
      VAR visitedIdx: the index of the set visited index of the IntegerArray
   @param
      leftBorder: the current left border of the array range
   @param
      rightBorder: the current right border of the array range
}
PROCEDURE BalanceTree;
  PROCEDURE AppendBalanced(VAR visited: IntegerArray; VAR personArray: PersonItemNodeArray;
VAR tree: PersonItemTree; VAR visitedIdx: LONGINT; leftBorder, rightBorder: LONGINT);
  VAR
    idx: LONGINT;
  BEGIN
    { If tree is not NIL and not all persons have been added }
    IF (NOT AllIndexesVisited(visited)) THEN BEGIN
      { Get middle index for given range }
      idx := GetMiddleIndex(leftBorder, rightBorder);
      IF (idx <> -1) THEN BEGIN
        Inc(visitedIdx);
        tree := personArray[idx];
        visited[visitedIdx] := idx;
        tree^.left := NIL;
        tree^.right := NIL;
        {
          WriteLn;
          WriteLn('----------------- Index -----------------');
          WriteLn('idx: ', idx);
          WriteLn('name: ', tree^.name);
          WriteLn('leftBorder: ', leftBorder);
          WriteLn('rightBorder: ', rightBorder);
          WriteLn('----------------- Index -------------------');

        WriteLn('get left middle'); }
        { Append left tree }
        AppendBalanced(visited, personArray, tree^.left, visitedIdx, leftBorder, (idx - 1));
        { WriteLn('get right middle'); }
        { Append right tree }
        AppendBalanced(visited, personArray, tree^.right, visitedIdx, (idx + 1), rightBorder);
      END;
    END;
  END;

VAR
  root: PersonItemTree;
  personArray: PersonItemNodeArray;
  visitedIdx, leftBorder, rightBorder: LONGINT;
  visited: IntegerArray;
  i: LONGINT;
BEGIN
  IF ((size <> 0) AND ((size MOD factor) = 0)) THEN BEGIN
    { Balance tree only if its not marked as balanced }
    visitedIdx := 0;
    leftBorder := 1;
    rightBorder := size;
    root := NIL;
    visited := InitVisitedArray;
    personArray := GetSortedPersonItemNodes;

    { Build up new tree }
    AppendBalanced(visited, personArray, root, visitedIdx, leftBorder, rightBorder);
    { Get the root node for the backed tree }
    tree := personArray[GetMiddleIndex(1, size)];
    {
      FOR i := 1 TO High(visited) DO BEGIN
        WriteLn('idx: ', visited[i]);
      END;
    }
    { Release dynamic arrays }
    SetLength(personArray, 0);
    SetLength(visited, 0);
```

```
  END;
END;
{
  Adds a PersonItem instance to the backed tree via an iterative implementation
  because there can be determined a linear way to the node to which the child has to be added.

  @param
      node: the PersonItem instance to the added to the backed tree
}
PROCEDURE AppendPersonItemNode(node: PersonItemNode);
VAR
  parent, subTree: PersonItemNode;
BEGIN

  { First item in the tree }
  IF (tree = NIL) THEN BEGIN
    tree := node;
    size := 1;
  END
  { Add item to end of tree }
  ELSE BEGIN
    parent := NIL;
    subTree := tree;
    WHILE (subTree <> NIL) DO BEGIN
      parent := subtree;
      IF (node^.name <= subTree^.name) THEN BEGIN
        subTree := subTree^.left;
      END
      ELSE BEGIN
        subTree := subTree^.right;
      END;
    END;

    { Add item on last node of the tree after the correct site has been determined }
    IF (node^.name < parent^.name) THEN BEGIN
      parent^.left := node;
    END
    ELSE BEGIN
      parent^.right := node;
    END;

    { Increase the size member, to keep the information about the size of the tree }
    size := size + 1;
  END;
END;

{
  Gets the PersonItemNode with the given name via an iterative implementation
  because there is a linear way to the searched node.

  @param
      name: the name of the person to find
  @return
      the found PersonItemNode or NIL if the person could not be found
}
FUNCTION FindPersonItemNodeForName(name: STRING): PersonItemNode;
VAR
  node: PersonItemNode;
  lowerName: STRING;
BEGIN
  node := tree;
  lowerName := LowerCase(name);
  WHILE ((node <> NIL) AND (node^.name <> lowerName)) DO BEGIN
    IF (lowerName < node^.name) THEN BEGIN
      node := node^.left;
    END
    ELSE BEGIN
      node := node^.right;
    END;
  END;
  FindPersonItemNodeForName := node;
END;

{
  Disposes the tree via an recursive implementation
  because all nodes have to be visited.
```

```
  @param
      VAR tree: the tree to be disposed.
}
PROCEDURE DisposeTree(VAR tree: PersonItemTree);
BEGIN
  IF (tree <> NIL) THEN BEGIN
    DisposeTree(tree^.left);
    DisposeTree(tree^.right);
    {
      WriteLn;
      WriteLn('-----------------------------------');
      WriteLn('Delete work items of: ', tree^.name);
      WriteLn('-----------------------------------');
    }
    DisposeWorkItemList(tree^.workItems);
    {
      WriteLn('Delete person: ', tree^.name);
    }
    Dispose(tree);
    tree := NIL;
  END
END;

{ ######################### Public Function/Procedure part ######################### }
{ Reset memory }
PROCEDURE Reset;
BEGIN
  DisposeTree(tree);
  size := 0;
END;

{ Creates a WorkEntry instance }
FUNCTION CreateWorkEntry(name, task: STRING; spendTime: TimeSpan): WorkEntry;
BEGIN
  CreateWorkEntry.name := LowerCase(name);
  CreateWorkEntry.task := LowerCase(task);
  CreateWorkEntry.spendTime := spendTime;
END;

{ Adds the work entry to the memory}
PROCEDURE AddWorkEntry(entry: WorkEntry; VAR error: ErrorCode);
VAR
  person: PersonItemNode;
BEGIN
  { Converts the time span to seconds, and will contain any error }
  entry.spendTime := TimeSpanToSeconds(entry.spendTime);
  { Invalid time span detected }
  IF (entry.spendTime.error <> '') THEN BEGIN
    error := INVALID_SPAN
  END
  { To less time span }
  ELSE IF (entry.spendTime.timeInSeconds < 60) THEN BEGIN
    error := TO_SHORT
  END
  { To much time }
  ELSE IF (entry.spendTime.timeInSeconds > (8 * 60 * 60)) THEN BEGIN
    error := TO_LONG
  END
  { valid to add entry }
  ELSE BEGIN
    person := FindPersonItemNodeForName(entry.name);
    { If a entry already exists for the given name }
    IF (person <> NIL) THEN BEGIN
      AddWorkItemSorted(person, entry);
    END
    { If new person has to be added }
    ELSE BEGIN
      AppendPersonItemNode(CreatePersonItemNode(entry));
      BalanceTree;
    END;
  END;
END;

{ Gets total work time }
PROCEDURE GetTotalWorkTimeForPerson(name: STRING; VAR spendTime: TimeSpan);
```

```
VAR
  node: PersonItemNode;
  sec, count: LONGINT;
BEGIN
  sec := 0;
  count := 0;

  node := FindPersonItemNodeForName(name);
  IF (node <> NIL) THEN BEGIN
    SumSpendTimeOfWorkItemList(node^.workItems, sec, count);
  END;

  spendTime := TimeSpanUnit.SecondsToTimeSpan(sec);
END;

{ Gets average of spend time }
PROCEDURE GetAverageWorkTimeForTask(task: STRING; VAR average: TimeSpan);
  {
    Gets the average work time for the task.

    @param
        VAR tree: the tree to be traversed
     @param
        task: the task to get average work time for
     @param
        VAR seconds: the summary of the seconds
     @param
        VAR count: the summary of found work entries
  }
  PROCEDURE GetAverageWorkTime(VAR tree: PersonItemTree; task: STRING; VAR seconds, count:
LONGINT);
  VAR
    items: WorkItemList;
  BEGIN
    IF (tree <> NIL) THEN BEGIN
      GetAverageWorkTime(tree^.left, task, seconds, count);
      GetAverageWorkTime(tree^.right, task, seconds, count);
      { Search for this task }
      items := FindWorkItemNodesForTask(tree, task);
      { Build sum over the Work Items }
      SumSpendTimeOfWorkItemList(items, seconds, count);
      { Dispose WorkItem List because it has been copied as interface spec says }
      DisposeWorkItemList(items);
    END
  END;

VAR
  seconds, count: LONGINT;
  subTree: PersonItemTree;
BEGIN
  seconds := 0;
  count := 0;
  average.timeInSeconds := 0;
  subTree := tree;

  GetAverageWorkTime(subTree, task, seconds, count);
  { Calculate the average if the task on working persons have been found }
  IF (seconds <> 0) THEN BEGIN
    seconds := (seconds DIV count);
  END;
  { Create time span out of seconds }
  average := TimeSpanUnit.SecondsToTimeSpan(seconds);
END;

{ Print persons for task to table }
PROCEDURE PrintPersonForTask(task: STRING);
  {
    Prints the person for the given task, sorted by their name.
    This is an recursive implemented because the nodes which are needed can not be identified
    by an contained attribute but by an attribute of one node of the hold WorkItemList.

    @param
        VAR tree: the tree to be traversed
    @param
        task: the task to search the persons for
    @param
```

```
        VAR the index of the to print person
    }
    PROCEDURE PrintPerson(VAR tree: PersonItemTree; task: STRING; VAR count: LONGINT);
    BEGIN
      IF (tree <> NIL) THEN BEGIN
        PrintPerson(tree^.left, task, count);
        IF (PersonHasTask(tree, task)) THEN BEGIN
          Inc(count);
          WriteLn('          ', count, ': ', tree^.name);
        END;
        PrintPerson(tree^.right, task, count);
      END
    END;

VAR
    count: LONGINT;
    subTree: PersonItemTree;
BEGIN
    count := 0;
    subTree := tree;

    WriteLn('################### Persons working on task ###################');
    WriteLn('Task    : ', task);
    WriteLn('Persons : ');

    { Print the persons for the task }
    PrintPerson(subTree, task, count);
    { Either no entries or task not found }
    IF (count = 0) THEN BEGIN
      WriteLn('          No work entries found for the task !!!');
    END;
    WriteLn('################### Persons working on task ###################');
END;

{ Print work summary for a person }
PROCEDURE PrintWorkSummaryForPerson(name: STRING);
VAR
    person: PersonItemNode;
    node: WorkItemNode;
BEGIN

    WriteLn('################### Work summary for person ###################');
    WriteLn('Person : ', name);

    { Only print result if a entry for the person exists }
    person := FindPersonItemNodeForName(name);
    IF (person <> NIL) THEN BEGIN
      node := person^.workItems;
      WHILE (node <> NIL) DO BEGIN
        WriteLn('         Task: ', node^.task:15, ' | Spend time: ',
TimeSpanUnit.TimeSpanToString(node^.spendTime));
        node := node^.next;
      END;
    END
    { Either no entries or person not found }
    ELSE BEGIN
      WriteLn('         No work entries found for the person !!!':10);
    END;
    WriteLn('################### Work summary for person ###################');
END;

{ Get busiest person }
FUNCTION BusiestPerson: STRING;
    {
      Gets the busiest person over all persons saved in the tree
      via an recursive implementation because all nodes in tree have to be visited.

      @param
          VAR tree: the tree to be traversed
      @param
          VAR oldSeconds: the seconds of the former recursive call
      @param
          VAR name: the name of the current busiest person
    }
    PROCEDURE Busiest(VAR tree: PersonItemTree; VAR oldSeconds: LONGINT; VAR name: STRING);
    VAR
```

```
    count, seconds: LONGINT;
  BEGIN
    seconds := 0;
    count := 0;

    IF (tree <> NIL) THEN BEGIN
      Busiest(tree^.left, oldSeconds, name);
      Busiest(tree^.right, oldSeconds, name);
      SumSpendTimeOfWorkItemList(tree^.workItems, seconds, count);
      if (seconds > oldSeconds) THEN BEGIN
        oldSeconds := seconds;
        name := tree^.name;
      END;
    END
  END;

VAR
  oldSeconds: LONGINT;
  name: STRING;
  subTree: PersonItemTree;
BEGIN
  oldSeconds := 0;
  name := '';
  subTree := tree;

  Busiest(subTree, oldSeconds, name);
  BusiestPerson := name;
END;

(*
{ No need to delete person in this version }
{ Removes a person }
FUNCTION DeletePerson(name: STRING): BOOLEAN;
VAR
  person: PersonItemNode;
BEGIN
  DeletePerson := false;
  person := FindPersonItemNodeForName(name);
  IF (person <> NIL) THEN BEGIN
    person^.left^.right := person^.right;
    person^.right^.left := person^.left;
    DisposeWorkItemList(person^.workItems);
    {
      WriteLn('Dispose Person: ', person^.name);
    }
    Dispose(person);
    DeletePerson := true;
  END;
END;
*)

{ Get work entry count }
FUNCTION GetTotalWorkEntryCount: LONGINT;
  {
    Gets the total work entry count of all person in the tree
    via an recursive implementation because all nodes in tree will have to be visited.

    @param
        VAR tree: the tree to traversed
    @param
        VAR count: the summary of the work entries
  }
  PROCEDURE GetTotalCount(VAR tree: PersonItemTree; VAR count: LONGINT);
  VAR
    workNode: WorkItemNode;
  BEGIN
    IF (tree <> NIL) THEN BEGIN
      GetTotalCount(tree^.left, count);
      GetTotalCount(tree^.right, count);
      workNode := tree^.workItems;
      WHILE (workNode <> NIL) DO BEGIN
        Inc(count);
        workNode := workNode^.next;
      END;
    END
  END;
```

```
VAR
  count: LONGINT;
  subTree: PersonItemTree;
BEGIN
  count := 0;
  subTree := tree;

  GetTotalCount(tree, count);
  GetTotalWorkEntryCount := count;
END;

BEGIN
  { Init tree by setting it to NIL }
  tree := NIL;
  { Call reset which set size to 0 }
  Reset;
END.
```

2.2 WorkManagementUnitTest

Folgend ist der Tests der WorkManagementUnitTest angeführt.

```
PROGRAM WorkManagementUnitTest;

USES WorkManagementUnit, TimeSpanUnit;

{
  Adds a entry and prints the added entry to the console
}
PROCEDURE AddEntry(name, task: STRING; span: TimeSpan);
VAR
  entry: WorkEntry;
  error: ErrorCode;
BEGIN
  error := NONE;
  entry := WorkManagementUnit.CreateWorkEntry(name, task, span);
  WorkManagementUnit.AddWorkEntry(entry, error);
  WriteLn('Error: ', error:13, 'Name: ', entry.name:10, ' | Task: ', entry.task:15, ' | Time:
', TimeSpanUnit.TimeSpanToString(entry.spendtime));
END;

{
  Tests GetTotalWorkTimeForPerson
}
PROCEDURE TestGetTotalWorkTimeForPerson;
VAR
  span: TimeSpan;
BEGIN
  WriteLn('---------------------- TestGetTotalWorkTimeForPerson ----------------------');

  WorkManagementUnit.Reset();
  span := TimeSpanUnit.CreateTimeSpan(0, 0, 0);
  WriteLn('No entries are present: ');
  WorkManagementUnit.GetTotalWorkTimeForPerson('Thomas', span);
  WriteLn('Total work time for Thomas: ', TimeSpanUnit.TimeSpanToString(span));
  WriteLn;

  WorkManagementUnit.Reset();
  AddEntry('Thomas' , 'Impl', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Thomas' , 'Spec', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Thomas' , 'Test', TimeSpanUnit.CreateTimeSpan(1, 0, 0));
  WorkManagementUnit.GetTotalWorkTimeForPerson('Maria', span);
  WriteLn('Total work time for Maria: ', TimeSpanUnit.TimeSpanToString(span));
  WriteLn;

  WorkManagementUnit.Reset();
  AddEntry('Thomas' , 'Impl', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Thomas' , 'Spec', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Thomas' , 'Test', TimeSpanUnit.CreateTimeSpan(1, 0, 0));
  AddEntry('Hannes' , 'Impl', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Hannes' , 'Spec', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Hannes' , 'Test', TimeSpanUnit.CreateTimeSpan(2, 2, 0));
  span := TimeSpanUnit.CreateTimeSpan(0, 0, 0);
```

```
  WorkManagementUnit.GetTotalWorkTimeForPerson('Thomas', span);
  WriteLn;

  WriteLn('Total work time for Thomas: ', TimeSpanUnit.TimeSpanToString(span));
  span := TimeSpanUnit.CreateTimeSpan(0, 0, 0);
  WorkManagementUnit.GetTotalWorkTimeForPerson('Hannes', span);
  WriteLn('Total work time for Hannes: ', TimeSpanUnit.TimeSpanToString(span));
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestGetTotalWorkTimeForPerson ----------------------');
  WriteLn;
END;

{
  Tests the AddWorkEntry
}
PROCEDURE TestAddWorkEntry;
BEGIN
  WriteLn('---------------------- TestAddWorkEntry BEGIN ----------------------');
  WorkManagementUnit.Reset();
  AddEntry('Thomas', 'Specification', TimeSpanUnit.CreateTimeSpan(0, 0, 1));
  AddEntry('THomas', 'Specification', TimeSpanUnit.CreateTimeSpan(8, 0, 1));
  AddEntry('THomas', 'Specification', TimeSpanUnit.CreateTimeSpan(1, 60, 60));
  AddEntry('HaNNes' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('HaNnes' , 'SpecificaTIon', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Thomas' , 'SpecificaTIon', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('THOmas' , 'SpecificaTIon', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Maria' , 'Specification', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  AddEntry('MAria' , 'SpecifICation', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  AddEntry('Julia' , 'SpecifICation', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  WriteLn;
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestAddWorkEntry END ----------------------');
  WriteLn;
END;

{
  Tests the procedure Rest
}
PROCEDURE TestReset;
BEGIN
  WriteLn('---------------------- TestReset ----------------------');
  WorkManagementUnit.Reset();
  AddEntry('Thomas', 'Impl', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  AddEntry('Thomas', 'Impl', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  AddEntry('Thomas', 'Spec', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  AddEntry('Thomas', 'Spec', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  AddEntry('Thomas', 'Test', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  AddEntry('Thomas', 'Test', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  AddEntry('Hannes' , 'Testing', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(8, 0, 0));
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestReset ----------------------');
  WriteLn;
END;

{
  Tests the GetAverageWorkTimeForTask
}
PROCEDURE TestGetAverageWorkTimeForTask;
VAR
  span: TImeSpan;
BEGIN
  WriteLn('---------------------- TestGetAverageWorkTimeForTask ----------------------');
  WorkManagementUnit.Reset();
  WriteLn('No entries are present: ');
  WorkManagementUnit.GetAverageWorkTimeForTask('Doku', span);
  WriteLn('AverageTime for Doku: ', TimeSpanUnit.TimeSpanToString(span));
  WriteLn;
  AddEntry('Thomas' , 'Impl', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Thomas' , 'Impl', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  WorkManagementUnit.GetAverageWorkTimeForTask('Doku', span);
  WriteLn('AverageTime for Doku: ', TimeSpanUnit.TimeSpanToString(span));
  WriteLn;

  WorkManagementUnit.Reset();
```

```
  AddEntry('Thomas' , 'Impl', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Thomas' , 'Impl', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Thomas' , 'Spec', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Thomas' , 'Spec', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Hannes' , 'Spec', TimeSpanUnit.CreateTimeSpan(1, 0, 0));
  AddEntry('Hannes' , 'Spec', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Hannes' , 'Impl', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Hannes' , 'Impl', TimeSpanUnit.CreateTimeSpan(0, 1, 0));

  WriteLn;
  span := TimeSpanUnit.CreateTimeSpan(0, 0, 0);
  WorkManagementUnit.GetAverageWorkTimeForTask('Impl', span);
  WriteLn;
  WriteLn('Average time for Impl: ', TimeSpanUnit.TimeSpanToString(span));

  span := TimeSpanUnit.createTimeSpan(0, 0, 0);
  WorkManagementUnit.GetAverageWorkTimeForTask('Spec', span);
  WriteLn;
  WriteLn('Average time for Spec: ', TimeSpanUnit.TimeSpanToString(span));

  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestGetAverageWorkTimeForTask ----------------------');
  WriteLn;
END;

{
  Tests the procedure PrintPersonForTask
}
PROCEDURE TestPrintPersonForTask;
BEGIN
  WorkManagementUnit.Reset();
  WriteLn('No entries are present: ');
  WorkManagementUnit.PrintPersonForTask('Implementation');

  WorkManagementUnit.Reset();
  AddEntry('Maria' , 'Test', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Thomas' , 'Test', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  WorkManagementUnit.PrintPersonForTask('Implementation');
  WriteLn;

  WriteLn('---------------------- TestPrintPersonForTask ----------------------');
  AddEntry('Maria' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Maria' , 'Specification', TimeSpanUnit.CreateTimeSpan(1, 0, 0));
  AddEntry('Markus' , 'Test', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(1, 1, 1));
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  AddEntry('Julia' , 'Doku', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  WriteLn;
  WorkManagementUnit.PrintPersonForTask('Implementation');
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestPrintPersonForTask ----------------------');
  WriteLn;
END;

{
  Test the procedure PrintWorkSummaryForPerson.
}
PROCEDURE TestPrintWorkSummaryForPerson;
BEGIN
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestPrintWorkSummaryForPerson ----------------------');
  WriteLn('No entries are present');
  WorkManagementUnit.PrintWorkSummaryForPerson('Thomas');
  WriteLn;
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(1, 1, 1));
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  WorkManagementUnit.PrintWorkSummaryForPerson('Thomas');
  WriteLn;

  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 2, 2));
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Thomas' , 'Specification', TimeSpanUnit.CreateTimeSpan(1, 1, 1));
  AddEntry('Thomas' , 'Test', TimeSpanUnit.CreateTimeSpan(2, 2, 2));
```

```
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 2, 0));
  AddEntry('Hannes' , 'Test', TimeSpanUnit.CreateTimeSpan(0, 1, 2));
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(1, 1, 1));
  AddEntry('Hannes' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 1));

  WorkManagementUnit.PrintWorkSummaryForPerson('Thomas');
  WriteLn;
  WorkManagementUnit.PrintWorkSummaryForPerson('Hannes');
  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestPrintWorkSummaryForPerson ----------------------');
  WriteLn;
END;

{
  Tests the function BusiestPerson
}
PROCEDURE TestBusiestPerson;
VAR
  name: STRING;
BEGIN
  WriteLn('---------------------- TestBusiestPerson ----------------------');
  WorkManagementUnit.Reset();
  WriteLn('No entries:');
  WriteLn('Buisiest person: ', WorkManagementUnit.BusiestPerson);

  WriteLn;
  WriteLn('single result:');
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 2));
  AddEntry('Maria' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  AddEntry('Maria' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  name := WorkManagementUnit.BusiestPerson;
  WriteLn('Buisiest person: ', name);

  WriteLn;
  WriteLn('multiple result:');
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 2));
  AddEntry('Maria' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 2));
  AddEntry('Maria' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 0));
  name := WorkManagementUnit.BusiestPerson;
  WriteLn('Buisiest person: ', name);

  WorkManagementUnit.Reset();
  WriteLn('---------------------- TestBusiestPerson ----------------------');
  WriteLn;
END;

{
  Tests the function GetToalWorkEntryCount
}
PROCEDURE TestGetToalWorkEntryCount;
VAR
  i, count: LONGINT;
  it: STRING;
BEGIN
  count := 10;
  WriteLn('---------------------- TestGetToalWorkEntryCount ----------------------');
  WorkManagementUnit.Reset();
  WriteLn('No entries:');
  WriteLn('Total work entry count: ', WorkManagementUnit.GetTotalWorkEntryCount);
  WriteLn;

  WriteLn('Adding ', count, ' different persons with each two work entry');
  FOR i := 1 TO count DO BEGIN
    Str(i, it);
    AddEntry((it + '-Thomas') , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
    AddEntry((it + '-Thomas') , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  END;
  WriteLn('Total work entry count: ', WorkManagementUnit.GetTotalWorkEntryCount);
  WriteLn('---------------------- TestGetToalWorkEntryCount ----------------------');
END;
```

```
(*
{
  Tests the function DeletePerson
}
PROCEDURE TestDeletePerson;
VAR
  result: BOOLEAN;
BEGIN
  WriteLn('---------------------- TestGetToalWorkEntryCount ----------------------');
  WorkManagementUnit.Reset();
  WriteLn('No entries:');
  WriteLn('Deleted: ', WorkManagementUnit.DeletePerson('thomas'));

  WriteLn;
  WorkManagementUnit.Reset();
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 2));
  AddEntry('Maria' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 2));
  AddEntry('Maria' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  WriteLn;
  WriteLn('Deleted: ', WorkManagementUnit.DeletePerson('hannes'));

  WriteLn;
  WorkManagementUnit.Reset();
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  AddEntry('Hannes' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 2));
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  AddEntry('Thomas' , 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 2));
  AddEntry('Maria' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 2));
  AddEntry('Maria' , 'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  WriteLn;
  result := WorkManagementUnit.DeletePerson('thomas');
  WriteLn;
  WriteLn('Deleted: ', result);
  WriteLn;
  WriteLn('Total work entry count: ', WorkManagementUnit.GetTotalWorkEntryCount);
  WriteLn;
  WorkManagementUnit.PrintWorkSummaryForPerson('maria');
  WriteLn;
  WorkManagementUnit.PrintWorkSummaryForPerson('hannes');

  WriteLn('---------------------- TestGetToalWorkEntryCount ----------------------');
END;
*)

{
  Tests the performance when work items are saved for one person
}
PROCEDURE TestPerformanceWorkEntry;
VAR
  i: LONGINT;
  it: STRING;
  error: ErrorCode;
BEGIN
  FOR i := 1 TO 10000 DO BEGIN
    Str(i, it);
    WorkManagementUnit.AddWorkEntry(WorkManagementUnit.CreateWorkEntry('Thomas',
'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 1)), error);
  END;
  WriteLn('Adding new task on last index');
  WorkManagementUnit.AddWorkEntry(WorkManagementUnit.CreateWorkEntry('Thomas',
'Specification', TimeSpanUnit.CreateTimeSpan(0, 1, 1)), error);
  { WorkManagementUnit.PrintWorkSummaryForPerson('Thomas'); }
  WriteLn('Total work entry count: ', WorkManagementUnit.GetTotalWorkEntryCount);
END;

{
  Tests the performance when multiple persons are saved with each one work item
}
PROCEDURE TestPerformancePersons;
VAR
  i: LONGINT;
  it: STRING;
BEGIN
  FOR i := 1 TO 20000 DO BEGIN
```

```
    Str(i, it);
    AddEntry((it + '-Thomas'), 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  END;
  WriteLn('Finished Insert');
  WriteLn('Total work entry count: ', WorkManagementUnit.GetTotalWorkEntryCount);
END;

{
  Tests the performance when multiple persons are saved with each 10 additonal work items
}
PROCEDURE TestPerformancePersonWorkEntry;
VAR
  i, j: LONGINT;
  it, it2: STRING;
BEGIN
  WorkManagementUnit.PrintPersonForTask('Implementation');
  FOR i := 1 TO 50000 DO BEGIN
    Str(i, it);
    AddEntry((it + '-Thomas'), 'Implementation', TimeSpanUnit.CreateTimeSpan(0, 1, 1));
  END;
  { WorkManagementUnit.PrintPersonForTask('Implementation'); }
  WriteLn('Finished Insert');
  WriteLn('Total work entry count: ', WorkManagementUnit.GetTotalWorkEntryCount);
END;

BEGIN
  (*
  { Test AddWorkEntry }
  TestAddWorkEntry;
  { Test TestReset }
  TestReset;
  { TestGetTotalWorkTimeForPerson }
  TestGetTotalWorkTimeForPerson;
  { Tests GetAverageWorkTimeForTask }
  TestGetAverageWorkTimeForTask;
  { Test for PrintWorkSummaryForPerson }
  TestPrintWorkSummaryForPerson;
  { Test PrintPersonForTimeTask }
  TestPrintPersonForTask;
  { Test TestBusiestPerson }
  TestBusiestPerson;
  { Test GetToalWorkEntryCount }
  TestGetToalWorkEntryCount;
  { Test Performance with work entries }
  TestPerformanceWorkEntry;
  {
    Test Performance with persons and work entries
    Set constant factor to 1 to se performance decrease.
  }
  TestPerformancePersonWorkEntry;
  *)

  (*
  No implemented for this version !!!!!!
  { Test DeletePerson }
  TestDeletePerson;
  *)
END.
```

3 Tests

Folgend sind die Tests für die WorkManagementUnit angeführt.

3.1 AddWorkEntry

Folgend sind Tests für die Prozedur AddWorkEntry angeführt.

Ungültige Arbeitseinträge:

```
Error: TO_SHORT     Name:     thomas | Task:   specification | Time: 0:0:1
Error: TO_LONG      Name:     thomas | Task:   specification | Time: 8:0:1
Error: INVALID_SPAN Name:     thomas | Task:   specification | Time: 1:60:60
```

Bei ungültigen Arbeitseinträgen (ungültige Zeitangaben) werden die entsprechenden Fehlermeldungen als Enumeration zurückgeliefert.

Gültige Arbeitseinträge:

```
Error: NONE         Name:     hannes | Task:   specification | Time: 0:1:0
Error: NONE         Name:     hannes | Task:   specification | Time: 0:1:0
Error: NONE         Name:     thomas | Task:   specification | Time: 0:1:0
Error: NONE         Name:     thomas | Task:   specification | Time: 0:1:0
Error: NONE         Name:      maria | Task:   specification | Time: 8:0:0
Error: NONE         Name:      maria | Task:   specification | Time: 8:0:0
Error: NONE         Name:      julia | Task:   specification | Time: 8:0:0

-----------------------------------
Delete work items of: julia
-----------------------------------
Dispose WorkItem:   specification | spendTime: 8:0:0
Delete person: julia

-----------------------------------
Delete work items of: maria
-----------------------------------
Dispose WorkItem:   specification | spendTime: 8:0:0
Dispose WorkItem:   specification | spendTime: 8:0:0
Delete person: maria

-----------------------------------
Delete work items of: thomas
-----------------------------------
Dispose WorkItem:   specification | spendTime: 0:1:0
Dispose WorkItem:   specification | spendTime: 0:1:0
Delete person: thomas

-----------------------------------
Delete work items of: hannes
-----------------------------------
Dispose WorkItem:   specification | spendTime: 0:1:0
Dispose WorkItem:   specification | spendTime: 0:1:0
Delete person: hannes
```

Wenn neue Personen hinzugefügt werden, so werden diese auch den Baum hinzugefügt. Sollte die Person bereits existieren, so wird der Arbeitseintrag der Person hinzugefügt. Die Korrektheit der Speicherung und ebenso des Löschens ist in der Konsolenausgabe ersichtlich.

3.2 Reset

Folgend sind die Tests der Prozedur Reset angeführt.

```
---------------------- TestReset -----------------------
Error: NONE         Name:     thomas | Task:            impl | Time: 8:0:0
Error: NONE         Name:     thomas | Task:            impl | Time: 8:0:0
Error: NONE         Name:     thomas | Task:            spec | Time: 8:0:0
Error: NONE         Name:     thomas | Task:            spec | Time: 8:0:0
Error: NONE         Name:     thomas | Task:            test | Time: 8:0:0
Error: NONE         Name:     thomas | Task:            test | Time: 8:0:0
Error: NONE         Name:     hannes | Task:   specification | Time: 8:0:0
Error: NONE         Name:     hannes | Task:         testing | Time: 8:0:0
Error: NONE         Name:     hannes | Task:  implementation | Time: 8:0:0

-----------------------------------
Delete work items of: hannes
-----------------------------------
Dispose WorkItem:  implementation | spendTime: 8:0:0
Dispose WorkItem:   specification | spendTime: 8:0:0
Dispose WorkItem:         testing | spendTime: 8:0:0
Delete person: hannes

-----------------------------------
Delete work items of: thomas
-----------------------------------
Dispose WorkItem:            impl | spendTime: 8:0:0
Dispose WorkItem:            impl | spendTime: 8:0:0
Dispose WorkItem:            spec | spendTime: 8:0:0
Dispose WorkItem:            spec | spendTime: 8:0:0
Dispose WorkItem:            test | spendTime: 8:0:0
Dispose WorkItem:            test | spendTime: 8:0:0
Delete person: thomas
---------------------- TestReset -----------------------
```

Bevor die Personen gelöscht werden, werden die Arbeitseinträge die von der Peron gehalten werden gelöscht.

3.3 GetTotalWorktimeForPerson

Folgend sind die Tests der Prozedur GetTotalWorktimeForPerson angeführt.

<u>Keine Einträge:</u>

```
No entries are present:
Total work time for Thomas: 0:0:0

Error: NONE         Name:     thomas | Task:            impl | Time: 0:1:0
Error: NONE         Name:     thomas | Task:            spec | Time: 0:2:0
Error: NONE         Name:     thomas | Task:            test | Time: 1:0:0
Total work time for Maria: 0:0:0
```

Sollten überhaupt keine Einträge vorhanden sein, oder die Person nicht gefunden werden, so wird eine entsprechende Meldung auf der Konsole ausgegeben.

<u>Einträge vorhanden:</u>

```
Error: NONE         Name:     thomas | Task:            impl | Time: 0:1:0
Error: NONE         Name:     thomas | Task:            spec | Time: 0:2:0
Error: NONE         Name:     thomas | Task:            test | Time: 1:0:0
Error: NONE         Name:     hannes | Task:            impl | Time: 0:2:0
Error: NONE         Name:     hannes | Task:            spec | Time: 0:2:0
Error: NONE         Name:     hannes | Task:            test | Time: 2:2:0

Total work time for Thomas: 1:3:0
Total work time for Hannes: 2:6:0
```

Wenn die Person gefunden wird, so wird die gesamte Arbeitszeit ausgegeben.

3.4 GetAverageWorkTimeForTask

Folgend sind die Tests der Prozedur GetAverageworkTimeForTask angeführt.

Keine Einträge:

```
No entries are present:
AverageTime for Doku: 0:0:0

Error: NONE         Name:     thomas | Task:            impl | Time: 0:2:0
Error: NONE         Name:     thomas | Task:            impl | Time: 0:1:0
AverageTime for Doku: 0:0:0
```

Wenn überhaupt keine Personen vorhanden sind, oder der Task nicht gefunden wurde, so wird dies über die Konsole ausgeben.

Einträge vorhanden:

```
Error: NONE         Name:     thomas | Task:            impl | Time: 0:2:0
Error: NONE         Name:     thomas | Task:            impl | Time: 0:1:0
Error: NONE         Name:     thomas | Task:            spec | Time: 0:2:0
Error: NONE         Name:     thomas | Task:            spec | Time: 0:2:0
Error: NONE         Name:     hannes | Task:            spec | Time: 1:0:0
Error: NONE         Name:     hannes | Task:            spec | Time: 0:2:0
Error: NONE         Name:     hannes | Task:            impl | Time: 0:1:0
Error: NONE         Name:     hannes | Task:            impl | Time: 0:1:0

Average time for Impl: 0:1:15

Average time for Spec: 0:16:30
```

Wenn Arbeitseinträge für den Task vorhanden sind, so wird der Durchschnitt der Arbeitszeit für den gewählten Task ausgeben. (Achtung Ganzzahlige Division)

3.5 PrintWorkSummaryForPerson

Folgend sind die Tests für die Prozedur PrintWorkSummaryForPerson angeführt.

Keine Einträge vorhanden:

```
No entries are present
#################### Work summary for person ####################
Person : Thomas
         No work entries found for the person !!!
#################### Work summary for person ####################

Error: NONE         Name:     hannes | Task:   specification | Time: 1:1:1
Error: NONE         Name:     hannes | Task:   specification | Time: 0:1:1
#################### Work summary for person ####################
Person : Thomas
         No work entries found for the person !!!
#################### Work summary for person ####################
```

Wenn überhaupt keine Personen vorhanden sind, oder die gewählte Person nicht gefunden werden kann so wird dies über die Konsole ausgegeben.

Einträge vorhanden:

```
Error: NONE         Name:     thomas | Task:  implementation | Time: 0:2:2
Error: NONE         Name:     thomas | Task:  implementation | Time: 0:1:0
Error: NONE         Name:     thomas | Task:   specification | Time: 1:1:1
Error: NONE         Name:     thomas | Task:            test | Time: 2:2:2
Error: NONE         Name:     hannes | Task:  implementation | Time: 0:2:0
Error: NONE         Name:     hannes | Task:            test | Time: 0:1:2
Error: NONE         Name:     hannes | Task:   specification | Time: 1:1:1
Error: NONE         Name:     hannes | Task:   specification | Time: 0:1:1
#################### Work summary for person ####################
Person : Thomas
         Task:  implementation | Spend time: 0:2:2
         Task:  implementation | Spend time: 0:1:0
         Task:   specification | Spend time: 1:1:1
         Task:            test | Spend time: 2:2:2
#################### Work summary for person ####################

#################### Work summary for person ####################
Person : Hannes
         Task:  implementation | Spend time: 0:2:0
         Task:   specification | Spend time: 1:1:1
         Task:   specification | Spend time: 0:1:1
         Task:   specification | Spend time: 1:1:1
         Task:   specification | Spend time: 0:1:1
         Task:            test | Spend time: 0:1:2
#################### Work summary for person ####################
```

Ist die gewählte Person vorhanden, so werden alle erfasste Arbeitseinträge angezeigt. Hierbei ist zu sehen, dass die Arbeitseinträge sortiert ausgeben werden. Dies ist möglich, da die Einträge sortiert gespeichert werden.

3.6 PrintPersonForTask

Folgend sind die Tests der Prozedur PrintPersonForTask angeführt.

Keine Einträge vorhanden:

```
#################### Persons working on task ####################
Task    : Implementation
Persons :
         No work entries found for the task !!!
#################### Persons working on task ####################
Error: NONE         Name:      maria | Task:            test | Time: 0:2:0
Error: NONE         Name:     thomas | Task:            test | Time: 0:1:0
#################### Persons working on task ####################
Task    : Implementation
Persons :
name: maria
name: thomas
         No work entries found for the task !!!
#################### Persons working on task ####################
```

Sollten keine Personen vorhanden sein, oder es keine gespeicherten Arbeitseinträge für den gewählten Task geben, so wird dies über die Konsole ausgeben.

Einträge vorhanden:

```
---------- TestPrintPersonForTask ----------
Error: NONE         Name:      maria | Task:  implementation | Time: 0:2:0
Error: NONE         Name:     thomas | Task:  implementation | Time: 0:1:0
Error: NONE         Name:      maria | Task:   specification | Time: 1:0:0
Error: NONE         Name:     markus | Task:            test | Time: 0:2:0
Error: NONE         Name:     hannes | Task:  implementation | Time: 0:2:0
Error: NONE         Name:     hannes | Task:  implementation | Time: 0:1:0
Error: NONE         Name:     hannes | Task:   specification | Time: 1:1:1
Error: NONE         Name:     hannes | Task:   specification | Time: 0:1:1
Error: NONE         Name:      julia | Task:            doku | Time: 0:2:0

#################### Persons working on task ####################
Task    : Implementation
Persons :
         1: hannes
         2: maria
         3: thomas
#################### Persons working on task ####################
---------- TestPrintPersonForTask ----------
```

Wenn es Einträge für den gewählten Task gibt, so werden alle Personen, die Einträge für diesen Task besitzen sortiert ausgegeben. Dies liegt an der rekursiven IN-ORDER Implementierung.

3.7 BusiestPerson

Folgend sind die Test der Funktion GetBusiestPerson angeführt.

Keine Einträge vorhanden:

```
No entries:
Buisiest person:
```

Sollten keine Personen vorhanden so wird ein Leerstring ausgeben.

Einträge vorahnden:

```
single result:
Error: NONE          Name:     thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:     thomas | Task:  implementation | Time: 0:1:2
Error: NONE          Name:      maria | Task:   specification | Time: 0:1:1
Error: NONE          Name:      maria | Task:   specification | Time: 0:1:1
Error: NONE          Name:     hannes | Task:  implementation | Time: 0:1:0
Error: NONE          Name:     hannes | Task:  implementation | Time: 0:1:0
Buisiest person: thomas

multiple result:
Error: NONE          Name:     thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:     thomas | Task:  implementation | Time: 0:1:2
Error: NONE          Name:      maria | Task:   specification | Time: 0:1:2
Error: NONE          Name:      maria | Task:   specification | Time: 0:1:1
Error: NONE          Name:     hannes | Task:  implementation | Time: 0:1:0
Error: NONE          Name:     hannes | Task:  implementation | Time: 0:1:0
Buisiest person: thomas
```

Sollte es ein eindeutiges Resultat geben, so wird diese Person zurückgeliefert.

Sollten mehrere Personen denselben Arbeitszeitaufwand besitzen, so wird die erste gefundene Person ausgeben und die weiteren ignoriert.

3.8 GetTotalWorkEntryCount

Folgend sind die Tests der Funktion GetToalWorkEntrycount angeführt.

Keine Einträge vorhanden:

```
Total work entry count: 0
```

Wenn keine Einträge vorhanden sind wird 0 als Resultat zurückgeliefert.

Einträge vorhanden:

```
Adding 10 different persons with each two work entry
Error: NONE          Name:   1-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   1-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   2-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   2-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   3-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   3-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   4-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   4-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   5-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   5-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   6-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   6-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   7-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   7-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   8-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   8-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   9-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:   9-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:  10-thomas | Task:  implementation | Time: 0:1:1
Error: NONE          Name:  10-thomas | Task:  implementation | Time: 0:1:1
Total work entry count: 20
```

Wenn Einträge vorhanden sind, so werden alle Arbeitseinträge über alle Personen hinweg gezählt und das Resultat wird zurückgeliefert.

3.9 Performance

Folgend sind die Tests angeführt, welche die Performance der WorkManagementUnit testen.

Arbeitseinträge für eine Person:

```
Adding new task on last index
Total work entry count: 10001
```

Wenn 10000 Arbeitseinträge für ein und dieselbe Person gespeichert werden und anschließend ein Arbeitseintrag der bei dem sortierten Einfügen am Ende der Liste angefügt wird, so ist kein Performanceeinbruch zu erkennen. Dies ist damit zu erklären, das über die Liste hierbei nur einmal zu iterieren ist. Testhalber kann auch die Prozedur GetWorkSummaryForPerson ausgeführt werden um zu sehen, das der letzte Eintrag am Ende der Liste gespeichert wird.

Große Anzahl an gespeicherten Personen:

Wenn eine große Anzahl an Personen gespeichert wird und der Baum bei jedem Insert ausbalanciert wird, dann ist zu sehen, dass je mehr Personen bereits im Baum vorhanden sind, desto mehr lässt die Performance nach. Daher wurde entschieden den Baum nur nach allen 100 gespeicherten Personen auszubalancieren. Es sei hier auf den Source der WormManangementUnitTest verweisen.

4 Diskussion

Folgend ist die Diskussion der Implementierung angeführt.

4.1 Sortierte Arbeitseinträge

In dieser Implementierung werden die Arbeitseinträge einer Person sortiert in einer einfach verketteten Liste gespeichert. Dies könnte bei einer sehr großen Liste zu Performanceeinbußen führen, aber der Test hat gezeigt, dass bei einer Liste von 10000 Einträgen keine nennenswerten Performanceeinbußen zu erkennen waren, da die Liste bei einem Insert nur einmal durchlaufen wird. Der Vorteil ist, dass die Arbeitseinträge sortiert ausgegeben werden können.

4.2 Rekursive vs Iterative Implementierung

Bei allen Prozeduren, die es nicht erfordern alle Elemente im Baum zu besuchen konnten iterativ implementiert werden, da es sich hierbei um lineare endrekursive Algorithmen handelt. Bei den Prozeduren, die erfordern jeden Knoten im Baum zu besuchen wurde ein rekursiver Ansatz gewählt, da es hier nicht einfach möglich ist eine iterative Lösung zu implementieren (lösbar mittels Stack), da es hier keinen linearen Weg zu einen bestimmten Knoten gibt. Diese rekursiven Implementierungen sind nicht lineare rekursive Algorithmen.

4.3 Baum balancieren

Es wurde entschieden den Baum nach jedem hundertsten eingefügten Knoten auszubalancieren, da ein Ausbalancieren nach jedem Knoten, bei einem bereits großen Baum, die Performance beeinflussen würde. Siehe Tests. Es ist zwar möglich das der Baum hierbei aus der Balance ist, aber bei lediglich 100 neu hinzugefügten Knoten ist dies als akzeptabel anzusehen.